Implementing Query Rewriting for Consistent Query Answering in Databases

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

Alexander Celle
P. Universidad Católica de Chile, Depto, de Ciencia de Computación, Santiago, Chile.
acelle@puc.cl

Abstract

For several reasons, databases may be inconsistent with respect to a set of integrity constraints. Those inconsistent states must be somehow resolved in order to be able to use the information stored in them. In some cases, data cleaning could be an approach to get rid of these inconsistencies. However, this may be a complex and nondeterministic process that may lead to the loss of potentially useful information. To avoid this, we present in this paper an alternative approach, which consists in computing only those answers to queries that are consistent with the integrity constraints by means of an algorithm called QUECA. Given a first order query Q, the algorithm generates a new query $QUECA(Q)$, such that, when posed to a database, its answers correspond to the answers to Q that are consistent with the given integrity constraints. We prove QUECA to be sound, terminating and complete for a class of integrity constraints that includes most of the constraints found in traditional relational database systems. Complexity issues are also addressed. Finally, we describe the implementation of the algorithm in XSB. We take advantage of the functionalities of XSB, as a deductive database with tabling capabilities, and the possibility of coupling it to a relational database systems.

1 Introduction

Databases may be inconsistent with respect to certain integrity constraints (ICs). This may be due to several reasons, among them: (a) The impossibility, for a non expert user, of expressing and having automatically maintained certain classes of ICs in commercial database management systems (DBMSs) (without appealing to complex procedures or triggers); (b) The presence of user constraints that impose a user semantics on data; but are not checked/maintained by the DBMS; (c) Legacy data on which new semantic constraints need to be imposed; (d) Integration of data from different sources. More classical reasons include poor design of databases and transactional applications.

In some cases, data cleaning could be an approach to resolving inconsistencies. However this may be a complex, nondeterministic process that may lead to the loss of potentially useful information. However, the information stored in the database remains relevant to the user and is most likely useful. Furthermore, as long as the distinction between consistent and inconsistent data can be made, there is no need to discard data. In other situations, e.g. when a particular user (without control on the database administration) wants to impose his/her own constraints on the database (or some views of it), the database cannot be

cleaned or repaired, and it becomes necessary to find an alternative approach to satisfy his/her requirements.

Example 1. Consider the inclusion dependency stating that a purchase must have a corresponding client: $\forall u, v\ (Purchase(u, v) \rightarrow Client(u))$. The following database instance r violates the IC:

Purchase	
c	e_1
d	e_2
d	e_1

Client
c

When repairing the database we might be tempted to remove all the purchases done by client d, but this data provides us with useful information about a client's behavior, no matter whether he is a valid client or not. □

An alternative to the traditional view of ICs as constraints on data or database states, that accommodates to the situations described above, consists in considering (some of the) ICs as constraints on query answers. When a database is queried, it would be desirable to obtain only those answers that are consistent with the ICs. For example, if we are in the presence of user constraints, the answers obtained should be those that are in semantic correspondence with the user's view of the world. In consequence, a promising alternative to restoring consistency is to keep the inconsistent data *in* the database and modify the queries in order to retrieve only consistent information. By using this kind of approach one can still use the inconsistent data for analysis (purchases of customer d in Example 1).

In (Arenas *et al.*, 1999) a precise notion of consistent answer to a query was given. In essence, given a set of ICs IC, a tuple answer $\bar{t}$ is a consistent answer to a query $Q(\bar{x})$ in a inconsistent relational database instance r wrt IC if $Q(\bar{t})$ becomes true in every instance that is a repair of r. A repair is an instance over the same schema and domain that makes IC true and can be obtained from r by a minimal set of insertions or deletions of tuples. This definition of consistent answer is a model-theoretic definition that uses the database repairs as an auxiliary concept. In consequence, it is not the idea to construct all possible minimal repairs and then query them in order to determine consistent answers; this could be impossible or too complex. Actually, there may be exponentially many repairs (Arenas *et al.*, 2003a). An alternative mechanism for *consistent query answering* (CQA) becomes necessary.

In this direction, an operator T_ω was presented in (Arenas *et al.*, 1999), which does not repair the database, but, given a query $Q(\bar{x})$, computes a modified query $\mathrm{T}_\omega(Q)(\bar{x})$ to be posed to the original database instance r, in such a way that its answers are consistent in the semantic sense, i.e. they are true in all possible repairs. The operator was proven to be sound, complete and terminating for interesting syntactic classes of queries and ICs (Arenas *et al.*, 1999). This query rewriting based approach to CQA is particularly appealing, because when the operator, applied to a first-order query[1], terminates, it produces a transformed query that is also first-order; and then it can be expressed as an SQL query with polynomial data complexity (Vardi, 1982; Kanellakis, 1990).

A direct implementation of the operator T_ω is complicated due to the vast amount of redundant computation it performs and to the problem of checking its semantic termination condition. In consequence, in this paper we address the problem of designing and implementing an alternative operator inspired by T_ω. The new operator is defined by a query rewriting algorithm called $QUECA$, for "QUErying Consistent Answers". This algorithm, given a first-order query Q, generates a new query $QUECA(Q)$, whose answers

[1] Aggregate queries are being treated in (Arenas *et al.*, 2003a).

in r are consistent with IC, but as opposed to T_ω, it guarantees termination, soundness and completeness for a larger class of ICs. Furthermore, the new operator is easier to implement, because it has a better control on the intermediate structures that are generated. Here we report on its implementation using XSB (Sagonas *et al.*, 1994; Rao *et al.* 1997), a powerful logic programming system, whose useful functionalities are particularly appropriate for the implementation of the new CQA algorithm. In our implementation, XSB interacts with the DBMS DB2 of IBM, which stores the relational database.

This paper is structured as follows. Section 2 gives some preliminaries and reviews the most relevant notions and techniques introduced in (Arenas *et al.*, 1999); in particular the features of the operator T_ω, emphasizing the difficulties around its implementation. Also the class of ICs that are considered in this paper is characterized. Section 3 presents the algorithms used to generate a query $QUECA(Q)$ corresponding to a first-order query Q. Next, Section 4 provides an analysis of the algorithms in terms of runtime complexity, termination, soundness and completeness. The class of queries covered by the algorithm is characterized in Section 5. Section 6 describes the main issues related to the implementation of *QUECA* in XSB. Finally, Section 7 draws some conclusions, describes related work and proposes future directions of research. Appendix A gives the proofs of the results stated in the main body of the paper. Appendix B gives details about our implementation of consistent query answering. More implementation details can be found in (Celle, 2000).

This paper improves and extends (Celle and Bertossi, 2000). It contains the theoretical results and the implementation details missing in that previous version. Most importantly, several other papers published since then have helped clarify the scope of the query rewriting based approach to consistent query answering. The results in this paper are now seen from that perspective and put in a broader context.

2 Preliminaries

2.1 Basic Notions

Given a relational database schema, $\mathcal{S}$, that includes a fixed infinite database domain, we can consider the first-order language, $\mathcal{L}(\mathcal{S})$, constructed using the predicates in $\mathcal{S}$. Integrity constraints can be expressed as sentences in $\mathcal{L}(\mathcal{S})$. A database instance r can be seen as a set of ground atoms of $\mathcal{L}(\mathcal{S})$, or, alternatively, as a first-order structure compatible $\mathcal{S}$. Given a fixed finite set, IC, of integrity constraints, a relational database instance is consistent if it satisfies IC, i.e. $r \models IC$. Otherwise, we say that r is inconsistent. We assume that the set IC is logically consistent in the sense that there is a database instance that satisfies IC.

Given a possibly inconsistent database instance r, a *repair* r' of r is a database instance (over the same schema) that satisfies IC and differs from r by a minimal set, under set inclusion, of inserted or deleted tuples (Arenas *et al.*, 1999). That is, the set $(r \setminus r') \cup (r' \setminus r)$ is minimal under set inclusion in the class of all the instances r' with the same schema as r that satisfy IC. Notice that built-in atoms are not included in this set difference, because built-ins have a fixed interpretation in all database instances.

A ground tuple $\bar{t}$ is a *consistent* answer to a query $Q(\bar{x})$ wrt IC denoted by $r \models_c Q(\bar{t})$, if for every repair r' of r, $r' \models Q(\bar{t})$, i.e. r' makes Q true when the tuple variable $\bar{x}$ is replaced by $\bar{t}$. In this sense, consistent answers to a query are those that are invariant under minimal ways of restoring consistency.

Example 2. Consider a distributors database. $Provider(u, v)$ means that product v is provided by u, and $Receives(u, v)$ that product v is received from provider u. The following ICs state that the products supplied by a provider are received from the provider (and

vice versa), and that a provider supplies only one product.

$$\forall uv\ (Provider(u,v) \rightarrow Receives(u,v)), \tag{1}$$

$$\forall uv\ (Receives(u,v) \rightarrow Provider(u,v)), \tag{2}$$

$$\forall uvz\ (Provider(u,v) \wedge Provider(u,z) \rightarrow v = z). \tag{3}$$

The following database instance r, which violates *IC*,

Provider	
a	b
a	c
d	e

Receives	
a	b
a	c
d	e

has two repairs:

r' :

Provider	
a	c
d	e

Receives	
a	c
d	e

and r'' :

Provider	
a	b
d	e

Receives	
a	b
d	e

Here, the only consistent answer to the query $Provider(u,v)$? in the database instance r is (d,e): $r \models_c Provider(d,e)$. The query $\exists v Receives(u,v)$ has a and d as consistent answers. □

2.2 The T_ω *Operator*

The query rewriting approach to CQA relies on the concept of *residue* introduced in the context of semantic query optimization (Chakravarthy *et al.*, 1990). The T_ω operator, introduced in (Arenas *et al.*, 1999) to compute consistent answers, is an iterative operator that computes a sequence of queries. At each step of the iteration, each of the literals in the query gets its residues appended to it by means of a conjunction. Residues capture the interaction between a literal and the ICs and can be obtained by resolution applied to a literal and the ICs. We illustrate this notion.

Example 3. Consider the (implicitly universally quantified) integrity constraint

$$bakery(Name, Address, Phone, Owner, Closes) \rightarrow store(Name, Address, Phone); \tag{4}$$

and, as a query, the literal $bakery(x,y,z,u,v)$. The residue supplied by (4) for this literal is $store(x,y,z)$, meaning that, when retrieving a *bakery* as an answer, there must also exist a related *store* if we want the answer to be consistent with the IC. In other words, the query has to be satisfied together with its residue. If the database is consistent, this will automatically happen. However, if it is not, we may impose the residue as an extra condition, making sure that at least the answers to the original queries are consistent wrt the ICs. So, we can consider the residue $store(x,y,z)$ as appended via a conjunction to the original query $bakery(x,y,z,u,v)$. Finally, notice that the residue $store(x,y,z)$ can be obtained by applying resolution to the query $bakery(x,y,z,u,v)$ and the clausal form of (4), namely $\neg bakery(Name, Address, Phone, Owner, Closes) \vee store(Name, Address, Phone)$.[2] □

Remark 1

[2] Unless otherwise stated, ICs will be written in clausal form.

We need to introduce some conventions for the rest of the paper. The IC in (4) has the clausal form

$$\forall xyzuv\ (\neg B(x, y, z, u, v) \vee S(x, y, z)). \tag{5}$$

Consider now the non ground literal $\neg S(w_1, w_2, w_3)$. Resolving this literal with (5), leaving the free variables in the literal, we obtain as its residue $\phi = \neg B(w_1, w_2, w_3, u, v)$. In this residue, the variables u and v are (sometimes implicitly) universally quantified as a result of the (also usually implicit) universal quantification in (5). Actually, the exact form of the literal with its residue appended is $\neg S(w_1, w_2, w_3) \wedge \forall uv B(w_1, w_2, w_3, u, v)$. In this case, the variables of residue ϕ are $Var(\phi) = \{w_1, w_2, w_3, u, v\}$, but its quantified variables are u, v. When this residue is resolved with other ICs, e.g. to find its residues in turn, the variables u, v are those that will be substituted to participate in the unification in a resolution step. In any case, sometimes we will explicitly write the universal quantifiers in a residue like ϕ, if this help understanding things better, but sometimes we will not. In the latter case, we could consider the quantified variables, in this case u and v, as some sort of *marked* variables. □

A detailed description of the residue generation process is given in Section 3. Residues are used by the query rewriting operator T. In general terms, given a query Q in conjunctive normal form), $\mathrm{T}_\omega(Q)$ is defined as possibly infinite sequence of queries $\mathrm{T}_\omega(Q) := \{\mathrm{T}_0(Q), \mathrm{T}_1(Q), \mathrm{T}_2(Q), \ldots\}$ which have to be simultaneously satisfied. T_{i+1} is obtained from T_i by appending -in conjunction- to each literal introduced in step T_i, its residues wrt the ICs. Since the residues are additional conditions that make the query stronger, it always holds $\mathrm{T}_{i+1}(Q) \Rightarrow \mathrm{T}_i(Q)$. However, if $\mathrm{T}_n(Q) \Rightarrow \mathrm{T}_i(Q)$ for all $i \geq n$, we say that n is a *finiteness point* and the computation can be stopped. In this case, the sequence is equivalent to a first-order formula, namely to T_n. We illustrate the application of this operator by means of an example.

Example 4. With the set of ICs in Example 2, we will show a computation of $\mathrm{T}_\omega(P(u, v))$, letting P stand for *Provider* and R for *Receives*. First, notice that the residue of $P(u, v)$ with respect to the IC (1), written as $\neg Provider(u, v) \vee Receives(u, v)$, is $Receives(u, v)$, which can be obtained resolving the query atom and the IC. Analogously, the residue of $P(u, v)$ wrt the IC (3) (there are no residues for this atom wrt to (2)), written as $\neg Provider(u, v) \vee \neg Provider(u, z) \vee v = z$, is $\neg Provider(u, z) \vee v = z$. At each step of the iteration, the residues of the appearing literals have to be appended. In consequence, the following sequence is generated:

$$\begin{aligned}
\mathrm{T}_0(P(u, v)) &= P(u, v).\\
\mathrm{T}_1(P(u, v)) &= P(u, v) \wedge (R(u, v) \wedge (\neg P(u, z) \vee v = z)).\\
\mathrm{T}_2(P(u, v)) &= P(u, v) \wedge ((R(u, v) \wedge P(u, v)) \wedge ((\neg P(u, z) \wedge \neg R(u, z)) \vee v = z)).\\
\mathrm{T}_3(P(u, v)) &= P(u, v) \wedge ((R(u, v) \wedge P(u, v) \wedge (R(u, v) \wedge (\neg P(u, w) \vee v = w))) \wedge\\
&\quad ((\neg P(u, z) \wedge \neg R(u, z) \wedge \neg P(u, z)) \vee v = z)).
\end{aligned}$$

T_3 and T_2 look quite different, however it can be checked (Arenas *et al.*, 1999) that T_2 logically implies T_3, so the latter does not impose any new conditions. In consequence, 2 is a finiteness point, and the modified original query is equivalent to $\mathrm{T}_2(P(u, v))$. This is the query to be posed to the original database, and its answers should be consistent answers to the original query $P(u, v)$.

It should be clear that, from the syntactic point of view, the iteration in this example does not terminate. Syntactic termination is guaranteed in (Arenas *et al.*, 1999) for acyclic ICs, and those in this example are not. Here we have a point of semantic termination instead, but this point is not easily detected if we do not run a separate test for implication. □

The operator T_ω is sound, and complete for generic binary ICs (see Definitions 1 and 2 below), but in (Arenas *et al.*, 1999) its semantic termination is only guaranteed for *uniform* ICs, where an IC in clausal form is uniform if every variable in a literal also appears in some other literal in the clause (the ICs in Example 4 are obviously uniform). Even for uniform constraints, this termination theorem does not tell us, in concrete situations, when the semantic termination point is reached; and we could keep computing syntactically different iterations for ever if we run a straightforward implementation based only on the definition of T.

An approach mentioned in (Arenas *et al.*, 1999) to detection of semantic termination consists in using an automated theorem prover, in this case Otter (McCune, 1994), to check logical equivalence (or implication) between consecutive steps. However, apart from undecidability issues, this methodology can be cumbersome to implement, due to the intrinsic complexities of automated theorem proving techniques, but also due to the *off-line* nature of this part of the process.

Needless to say that when thinking of a possible implementation of consistent query answering based on the operator T, the termination issue is critical, and the problem has to be properly addressed, at least for some classes of ICs. In this direction, this paper deals with the problems of modifying the query rewriting approach to CQA, and implementing this modified version, in such a way that stronger completeness and termination results can be guaranteed for interesting and useful classes of ICs; not yet covered by the results for T_ω.

The nice properties of T_ω in terms of soundness and completeness are preserved under the modified version. In order to achieve all these goals, we need to add a stronger termination property which makes the new mechanism easier to implement. The basic approach involves a more careful analysis of the iteration process described above, with the purpose of identifying stronger syntactic conditions, that, when satisfied, guarantee semantically correct results.

2.3 Integrity Constraints

We will only consider *universal* constraints, which can be expressed in the *standard format* (Arenas *et al.*, 1999):

$$\forall(\bigvee_{i=1}^{m} P_i(\bar{x}_i) \vee \bigvee_{i=1}^{n} \neg Q_i(\bar{y}_i) \vee \psi),$$

where $\forall$ represents the universal closure of the formula, $\bar{x}_i$, $\bar{y}_i$ are tuples of variables, the $P_i(\bar{x}_i)$'s and $Q_i(\bar{y}_i)$'s are atomic formulas based on the schema predicates that do not contain constants, and ψ is a formula that mentions only built–in predicates (involving variables and, possibly constants). Notice that in these ICs, constants, if needed, can be pushed into ψ. The equality predicate is allowed in ψ. For example, the ICs in Example 4 are represented in the standard format.

Because of implementation issues, and departing from (Arenas *et al.*, 1999), we will negate the ICs in standard format, and by doing so, we represent ICs as denials. In consequence, we will consider as ICs only *range restricted* formulas (Nicolas, 1982) of the form

$$\leftarrow l_1 \wedge \cdots \wedge l_n, \tag{6}$$

where each l_i is a literal, i.e. an atom or the negation of an atom (a negative literal), and variables are assumed to be universally quantified over the whole formula. The range restriction requires that each variable appearing in a negative literal also appears in a positive literal. Furthermore, we expect that ICs do not generate or represent any specific data. In this sense, they are *generic*. They do not entail the presence or absence in/from the database of any specific piece of data. This idea is captured in the following definition.

Definition 1 (*Generic Constraints*)
A set of integrity constraints, IC, is *generic* if there is no ground literal L, such that $IC \nvDash L(\bar{a})$. □

Good properties of our rewriting approach will be ensured in the case of binary ICs.

Definition 2 (*Binary Integrity Constraints*)
A *binary integrity constraint* (BIC) is a range restricted denial of the form (6), that contains at most two database literals, that is, it is of the form, $\forall\ (\leftarrow l_1(\bar{x}_1) \wedge l_2(\bar{x}_2) \wedge \psi(\bar{x}))$. □

Usually ICs are generic. BICs account for an important class of integrity constraints found in relational databases. In this class we find functional dependencies, full inclusion dependencies and symmetry constraints. Furthermore, as a particular case of BICs, we find the *unary integrity constraints* (UICs), which have just one database literal plus possibly a formula with built-in predicates. UICs include domain and range constraints. In consequence, by considering generic BICs we are covering most of the static integrity constraints found in traditional relational databases. Outside this class we find (existential) referential ICs of the form $\forall\bar{x}\exists y\ (P(\bar{x}) \rightarrow Q(\bar{x}, y))$, transitivity constraints, and possibly other constraints that might be better expressed as rules or views at the application layer.

3 Query Generation for Consistent Answers

As mentioned before, residues show the logical interaction between an integrity constraint and a literal name, i.e. an expression of the form P or $\neg P$, where P is a database predicate. A literal name which does not appear in any constraint does not have any residues associated to it, i.e. there are no semantic restrictions on that literal. Similarly, a literal that appears more than once in an IC or in a set of ICs, may have several residues, which may or may not be redundant (see Definition 3 below).

To systematically calculate the residues of literal names of a database schema with ICs, we will introduce the algorithm *Res* (in Algorithm 1). Because only literal names appearing in an integrity constraint generate residues, the algorithm will only be applied to them, and not to every relation in the schema.

Once all the residues associated to a literal name appearing in IC have been calculated, it is possible to apply our second algorithm, *QUECA*, that generates the queries for CQA based on the residues that have been already computed. We will show later on how this algorithm differs from the operator T_ω presented in (Arenas *et al.*, 1999) in operational terms, but also in terms of termination and completeness, and the necessary conditions for these properties.

3.1 Residue Calculation

Residues are calculated for every literal name appearing in an integrity constraint. In order to do this, we first build a list, L_{IC}, of ICs and the list L_P of the distinct literals appearing in IC. The elements of L_{IC} will only include the bodies of the ICs, which have been previously represented in the form (6). That is, given the set of ICs, IC, we build a list, L_{IC}, whose elements are of the form $l_1 \wedge \ldots \wedge l_n$, where $\forall(\leftarrow l_1 \wedge \ldots \wedge l_n) \in IC$. It should be noted that when we negate a member of L_{IC}, we obtain a clause.

Example 5. The set IC of ICs in Example 2 expressed in the form (6) contains:

$$\leftarrow P(u,v) \wedge \neg R(u,v); \quad \Leftarrow \neg P(u,v) \wedge R(u,v); \quad \leftarrow P(u,y) \wedge P(u,z) \wedge y \neq z.$$

In this case, $L_{IC} = [P(u,v) \wedge \neg R(u,v), \neg P(u,v) \wedge R(u,v), P(u,v) \wedge P(u,z) \wedge y \neq z]$, and $L_P = [P(u,v), R(u,v), \neg P(u,v), \neg R(u,v)]$. Notice that literals that differ only in their variables are considered to be the same. □

Next, when computing the residues for a literal $l \in L_P$ wrt all the $ic \in L_{IC}$, it is checked if the new candidate residue obtained from an IC is redundant or not. Only in the latter case it is appended. The systematic procedure to obtain residues, *Res*, used in our implementation is described in Algorithm 1.

Algorithm 1 *Res* computes *residues*(l)

Require: Set of integrity constraints IC in denial form.

Ensure: For each literal l, *residues*(l) is a formula in CNF that contains all the residues associated to l.

1: Create list L_{IC} of integrity constraint bodies and a list L_P of distinct literals in L_{IC}.
2: **for all** $l \in L_P$ **do**
3: $i = 1$
4: **for all** $ic \in L_{IC}$ **do**
5: **for** each occurrence of l in ic **do**
6: delete l from $ic \mapsto \overline{ic}$
7: negate $\overline{ic}$ {Now $\overline{ic}$ is in clausal form}
8: $residue_i(l) := \overline{ic}$
9: $i := i + 1$
10: **end for**
11: **end for**
12: $n(l) := i$ {the number of residues associated to l}
13: **end for** (*)
14: **for all** $l \in L_P$ **do**
15: $residues(l) := \varnothing$
16: **for all** $i := 1$ to $n(l)$ **do**
17: **if** $residue_i(l)$ is not redundant **then**
18: $residues(l) := residues(l) \wedge residue_i(l)$
19: **end if**
20: **end for**
21: **end for**

Example 6. (example 5 continued) Applying *Res* up to the line marked with (*) to $l = P(u, v)$ and every member of L_{IC}, we would obtain one residue for each occurrence of P: $residue_1(P(u, v)) := R(u, v)$, $residue_2(P(u, v)) := \neg P(u, z) \vee v = z$ and $residue_3(P(u, v)) := \neg P(u, w) \vee w = v$. In this case, the first IC in the list contributes with one residue, and the third one, with two. At this point there could be already redundant residues (see Definition 3). □

Finally, a conjunction of all the residues associated to a given $l \in L_P$ is created and denoted by *residues*(l). In this process (steps below (*) in Algorithm 1), we take care of eliminating redundant residues as we build the conjunction. This elimination is done in order to reduce complexity in the following phase (*QUECA*).

Definition 3 (*Residue Redundancy*)
Let $R \wedge \phi$ be a conjunction of residues associated to a literal l, where R is a clause and ϕ a conjunction of clauses. We say that R is *redundant* in $R \wedge \phi$ if there exists a clause $R' \in \phi$

and a substitution $\theta : (Var(R') \setminus Var(l)) \rightarrow (Var(R) \setminus Var(l))$, such that $R'\theta \equiv R$. Here, $Var(E)$, etc., denotes the set of all (quantified or non-quantified) variables in the expression E. □

Note that, in the definition above, if R is redundant in $R \wedge \phi$, then $R \wedge \phi$ is logically equivalent to ϕ. The procedure for eliminating redundant residues is invoked in steps below (*) in Algorithm 1, but is not explicitly shown there. It corresponds to the subsumption algorithm that is based on unification and presented in (Chakravarthy *et al.*, 1990).

Example 7. (example 6 continued) By Definition 3, we have that $residue_3(P(u, v))$ is a redundant residue, because there exists a substitution $\theta\colon\ z \mapsto w$, such that $residue_2(P(u, v))\theta = residue_3(P(u, v))$. Thus, we have $residues(P(u, v)) = (R(u, v)) \wedge (\neg P(u, z) \vee v = z)$. □

Notice that Definition 3 does not allow to detect *all* redundancies, but only those of syntactic nature expressed in its condition. For example, if we had the following residues for $R(x)$: $residue_1(R(x)) = P(x) \vee x > 100$ and $residue_2(R(x)) = P(x) \vee x > 50$, the fact that $residue_1$ includes the information in $residue_2$, making the latter semantically redundant, would not be detected. This is most likely to occur when ICs are already redundant, a situation that could be possibly avoided in a design phase, before applying the algorithm.

As shown in Example 7, functional dependencies are a common case of ICs which generate redundant residues according to Definition 3. The reason why residue redundancy is not treated further, e.g. covering semantic redundancy, is due to the complexity of implementation, which could be far higher than the performance improvement we could get at the next stage (*QUECA*). Besides, residue redundancy can become such a large subject that it would deviate the central point of attention of this work.

Example 8. Finally, continuing with the application of Algorithm 1 to the set IC presented in Example 5, we obtain:
$residues(P(u, v)) = (R(u, v)) \wedge (\neg P(u, z) \vee v = z)$, $\quad residues(\neg P(u, v)) = (\neg R(u, v))$,
$residues(R(u, v)) = P(u, v)$, $\quad residues(\neg R(u, v)) = \neg P(u, v)$. □

3.2 Query Rewriting (QUECA)

In this section we will concentrate on the computation of rewritings of a queries that are database literals. Other queries will be considered in Section 5. We will assume that the residues of literals appearing in the ICs (written as denials) have been computed (other literals do not have any residues).

Given a query Q, we will generate the query, $QUECA(Q)$, which should deliver consistent answers from a consistent or inconsistent database. This query only differs from Q when Q has residues, so $QUECA(Q)$ should be only executed for queries whose corresponding literal name appears in IC (in denial form). So, for example, if the query is $Q(x)\colon R(x, a)$ (where a is a constant), we should check if the literal $R(x, y)$ (or, say $R(u, v)$, with u, v variables, it does not matter) has residues.

Initially, the query $QUECA(Q)$ is equal to Q, plus a list of pending residues which are the residues associated to Q calculated by the *Res* algorithm presented in the previous section. Since the residues for a literal are in CNF (see e.g. Example 7), we will treat the conjunction as a list and every clause in it as an element of a list. These residues are not yet part of the query, they form a list of pending clauses that must be resolved via some condition if they should belong to the query. Informally speaking, this condition checks if they add new information to it or not. If they do not, they are discarded; but if they do, they must be added to the query and their residues appended at the end of the residue list. This procedure is iterated until no residues are left to resolve, i.e. either we run out of residues or they have all been discarded. This procedure may not always terminate.

Example 9. Consider the following hypothetical pairs of queries and residues:

Query:			*Residues*:	
	1.	$S(u)$		$S(u)$
	2.	$M(u)$		$N(u)$
	3.	$P(u, v)$		$\forall z\ (P(u, v) \vee \neg Q(u, z))$.

In the first case, the residue can be immediately discarded, because clearly it adds no new information to the query. However, in the second and third cases, the residues must be added to the corresponding query, and their residues to the Pending Residue List. So we would have:

Query:			*Residues*:	
	1.	$S(u)$		$\varnothing$
	2.	$M(u) \wedge N(u)$		$\mathit{residues}(N(u))$
	3.	$P(u, v) \wedge \forall z\ (P(u, v) \vee \neg Q(u, z))$		$\mathit{residues}(P(u, v) \vee \neg Q(u, z))$.

The residues that were added to the respective Pending Residue Lists are only mentioned (e.g. $\mathit{residues}(N(u))$), but not explicitly given to avoid cluttering the example. □

Everything works fine when only conjunctions are involved, e.g. case 2 in Example 9, because determining if a residue should be part of the query or not is easy. However, residues may be more complex clauses, e.g. case 3 in Example 9, so we must somehow deal with disjunction. Actually, in Section 3.1 we defined $\mathit{residues}(l)$, where l is a literal, but not when it is a more complex formula, like case 3. in Example 9.

We confront this problem by keeping conjunctions together, i.e. working in disjunctive normal form (DNF). To do so, when a clausal residue adds new information to a query, we make as many copies of the query as literals are present in the residue we are adding, and append to each of them exactly one of the literals in the residue. The pending residue list of each of these new copies must then be the existing list, minus the *resolved* residue (the one being considered), plus the residues coming from the newly appended literal. We shall informally call this a *split* operation. These copies with the added residues, connected together by disjunctions, would constitute the final query $QUECA(Q)$.

Example 10. (example 9 continued) In the third case in Example 9 we would then have

Query:			*Residues*:	
3.	E_1:	$P(u, v) \wedge P(u, v)$	R_1:	$\mathit{residues}(P(u, v))$
	E_2:	$P(u, v) \wedge \neg Q(u, z)$	R_2:	$\mathit{residues}(\neg Q(u, z))$.

Here each E_i is a disjunction free formula, and the list of residues for the original query is $\mathit{Residues} = R_1, R_2$. It should be noted that each R_i is associated to a corresponding E_i. If the literals $P(u, v)$ and $\neg Q(u, z)$ do not have any residues, we will have $QUECA(Q) = \forall z(E_1 \vee E_2)$. □

We can see that each of the E_i's will, in the end, form the final query $QUECA(Q)$. They will be connected by means of disjunctions, once all the pending residues have been resolved (more about this later).

We must, somehow, keep track of the correspondence between the mentioned copies (E_i's in Example 10) and their associated Pending Residue List (R_i's in Example 10). For this purpose, we will introduce a new notation that will help to keep track of the residues involved in building each E_i. Furthermore, this notation should not only include the literals in E_i and its associated Pending Residue List R_i, but, in order to avoid inserting a residue whose information was already inserted earlier, it should also "remember" the last residue that provoked one of these split operations. For these purposes we define a *temporary query unit* (TQU).

Definition 4 (*Temporary Qury Unit*)

A *temporary query unit* (TQU), $D : E \bullet R$, consists of a set of clauses D, a conjunction of literals E and a conjunction of residues R. A disjunction of temporary query units is called a *disjunctive* TQU ($DTQU$). □

The symbols ':' and '•' in a TQU are used only for separating its three elements. D represents the last residues involved in building E (those that must be "remembered") and R is the conjunction of residues $\phi_1 \wedge \cdots \wedge \phi_n$ yet to be resolved, associated to that particular E. We should note that all variables coming from a residue appear universally quantified in D and E. Also, both symbols have higher precedence that any other connective. We should also note that a TQU is a DTQU in particular. In the following example, we illustrate how a TQU is formed, the composition of the DTQU and what will constitute the final query $QUECA(Q)$.

Example 11. (example 10 continued) Using the new notation for the case in Example 10, we would have:

$$DTQU = \begin{array}{l} \overbrace{P(u,v) \vee \neg Q(u,z)}^{D_1} : \overbrace{P(u,v) \wedge P(u,v)}^{E_1} \bullet \overbrace{residues(P(u,v))}^{R_1} \\ \qquad\qquad\qquad\qquad \vee \\ \underbrace{P(u,v) \vee \neg Q(u,z)}_{D_2} : \underbrace{P(u,v) \wedge \forall z \neg Q(u,z)}_{E_2} \bullet \underbrace{residues(\neg Q(u,z))}_{R_2}, \end{array}$$

which we can also write as $DTQU = TQU_1 \vee TQU_2$. The final query should eventually be formed by the E_i's, i.e. $QUECA(Q) = \bigvee_i E_i$, once all the residues have been resolved. □

As may be expected, the remaining Pending Residue Lists, R_1 and R_2 in Example 11 (and R_i's in general), must again be resolved against their corresponding E_i's and D_i's in an iterative fashion. This iteration will stop when we run out of pending residues, that is, when $R_i = \varnothing$ for every i. The naive way to detect this is by observing at which point the • symbol reaches the right end of a given TQU. In this case, all its pending residues have been resolved.

The critical step is then, determining when a residue ϕ should be added to the query and when its information is already in it, i.e. it should be discarded. It is easy to see that ϕ can be discarded whenever $D \models \phi$ or $E \models \phi$. If either condition is not satisfied, the residue must be included in the query. Actually, we will see that sometimes only part of the residue must be included.

In Example 11, we have that $D_1 \vDash residues(P(u,v))$ (see the residues for the third case in Example 9), thus they can be discarded and the iteration would have ended for TQU_1. This form of redundancy is semantic issue that we want to capture by syntactic means, appealing to unification. In our case, we will define a sort of one way unification in which only certain types of variables will be involved, the new variables in a TQU and the free variables in a residue.

Definition 5 (*New and Free Variables*)
(a) A *new variable* in a $TQU = D : E \bullet R$ associated to a query Q is one that belongs to $newVar(TQU) := Var(E) \setminus Var(Q)$ and is universally quantified.
(b) Given a $TQU = D : E \bullet R$, a *free variable in a residue* $\phi \in R$ is one that belongs to $freeVar(\phi) := Var(\phi) \setminus Var(E)$ and is universally quantified. □

Because D in a TQU consists of a recently resolved residue, it also behaves as one and has *Free Variables* in the sense of Definition 5(b). For instance, in Example 11, we have $newVar(TQU_2) = \{z\}$ and $freeVar(D_1) = \{z\}$. From these definitions it is clear that we can substitute a *freeVar* for any other variable, because they occur nowhere else in that residue.

Having identified the variables that we will use in the unification process, we can now

formally define the meaning of *the information of a residue already in a TQU*. This notion will enable us to determine when a residue can be discarded or not.

Definition 6 (*Information of a Residue*)
We say the information of a residue $\phi = l_1 \vee \cdots \vee l_n$ is already in a $TQU = D : E \bullet R$, and we write $\phi \,\widetilde{\in}\, D : E$, whenever $\phi \in R$, and there exists a substitution $\theta : \mathit{freeVar}(\phi) \to \mathit{newVar}(TQU) \cup \mathit{freeVar}(D)$, such that $\phi\theta \in D$, or, for all i, $l_i\theta \in E$. Otherwise, we write $\phi \,\widetilde{\notin}\, D : E$. However, in case only some $l_i\theta \in E$, we say the information of the residue is partially in TQU, and we write $\phi \;{}_P\widetilde{\in}_\theta\; D : E$ □

Example 12. Consider the third case in Example 9:

$$\mathit{Query}: \quad 3. \quad P(u,v) \qquad \mathit{Residues}: \quad \forall z\ (P(u,v) \vee \neg Q(u,z)),$$

from which we can build the TQU $TQU = \varnothing : P(u,v) \bullet \forall z\ (P(u,v) \vee \neg Q(u,z))$.

Now, we must check if the information of this first, clausal residue is already in $\varnothing : P(u,v)$ or not. Clearly we have that $\forall z\ (P(u,v) \vee \neg Q(u,z)) \,\widetilde{\notin}\, [\varnothing{:}P(u,v)]$.[3] However, we do have that $\forall z\ (P(u,v) \vee \neg Q(u,z)) \;{}_P\widetilde{\in}_\theta\; [\varnothing{:}P(u,v)]$ for $\theta = \varepsilon$ (the identity, i.e. $P(u,v)\theta \in P(u,v)$). Thus, the literal $P(u,v)$ in the residue can be discarded (it does not contribute with any new information according to Definition 6, in particular, it does not contribute with new residues), so we keep only one instance of $P(u,v)$. The other member of the residue, $\forall z \neg Q(u,z)$, is appended to a copy of $P(u,v)$, and its residues appended to the Pending Residue List. If, for illustration purposes, we assume that $\mathit{residues}(\neg Q(x,y)) = P(x,y)$, we would have:

$$\begin{aligned} DTQU = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,v) \vee \neg Q(u,z) : P(u,v) \wedge \forall z \neg Q(u,z) \bullet P(u,z). \end{aligned}$$

We can see that the iteration has reached an end for the first member of $DTQU$. On the other hand, iterations must continue for the second member of the disjunction. We have that $P(u,z) \,\widetilde{\notin}\, [P(u,v) \vee \neg Q(u,z) : P(u,v) \wedge \forall z \neg Q(u,z)]$, so the residue must be added to $P(u,v) \wedge \forall z \neg Q(u,z)$, and its residues to the Pending Residue List. So we have:

$$\begin{aligned} DTQU = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,z) : P(u,v) \wedge \forall z(\neg Q(u,z) \wedge P(u,z)) \bullet \forall w(P(u,z) \vee \neg Q(u,w)). \end{aligned}$$

Now we have that $\forall w(P(u,z) \vee \neg Q(u,w)) \,\widetilde{\in}\, [P(u,z) : P(u,v) \wedge \forall z(\neg Q(u,z) \wedge P(u,z))]$ for $\theta : w \to z$. Thus the residue is discarded and iteration terminates with:

$$\begin{aligned} DTQU = \quad & P(u,v) \vee \neg Q(u,z) : P(u,v) \bullet \varnothing \\ & \qquad\qquad \vee \\ & P(u,z) : P(u,v) \wedge \forall z(\neg Q(u,z) \wedge P(u,z)) \bullet \varnothing. \end{aligned}$$

At this point, all the residues have been resolved and we have that the final query for consistent answers is $QUECA(P(u,v)) = P(u,v) \vee (P(u,v) \wedge \forall z(\neg Q(u,z) \wedge P(u,z))$ □

The general procedure illustrated in Example 12 can be summarized as follows. When we want to decide whether to add a residue $\phi = l_1 \vee \cdots \vee l_m$ to a query, then $\phi \,\widetilde{\in}\, D : E$ is checked, and in the positive case ϕ is discarded. Otherwise, it must be added to E and one of the split operations must take place. However, if $\phi \;{}_P\widetilde{\in}_\theta\; D : E$, then we must keep a copy of $D : E \bullet R$, adding ϕ to D; and for all the cases in which $l_i\theta \notin E$, $l_i\theta$ must be appended to a copy E_i of E, its residues must be added at the end of a copy R_i of R and D_i must be replaced by ϕ. The procedure is completely described in Algorithm 2.

[3] We use square brackets around the first two components of a TQU just for readability.

Algorithm 2 Generate a QUEry for Consistent Answers for a literal l: $QUECA(l)$

Require: Algorithm 1 has been executed.

1: $QUECA(l) := \varnothing$
2: $DTQU := \varnothing : l \bullet residues(l)$
3: **while** $DTQU \neq \varnothing$ **do**
4: select(extract) first TQU from $DTQU \mapsto (D\colon E \bullet R)$
5: **if** $R = \varnothing$ **then**
6: $QUECA(l) := QUECA(l) \vee E$
7: **else**
8: select(extract) first residue(clause) from $R \mapsto \phi$ $\{\phi = l_1 \vee \cdots \vee l_m\ \}$
9: **if** $\phi \,\widetilde{\in}\, D : E$ **then**
10: $DTQU = D\colon E \bullet R \vee DTQU$
11: **else**
12: **if** $\phi \;{}_P\widetilde{\in}_\theta\; D : E$ **then**
13: $append(D, \phi) \mapsto D_0$
14: $E_0 := E$
15: $R_0 := R$
16: **else**
17: $\theta = \varepsilon$ (identity)
18: **end if**
19: **for all** $i \in [1, m]$ **do**
20: **if** $l_i\theta \notin E$ **then**
21: $D_i := \phi$
22: $E_i := E \wedge l_i\theta$
23: $R_i := R \wedge residues(l_i\theta)$
24: **else**
25: Do nothing
26: **end if**
27: **end for**
28: $DTQU := \bigvee_{i=0}^{m}(D_i\colon E_i \bullet R_i) \vee DTQU$
29: **end if**
30: **end if**
31: **end while**

Example 13. (example 4 continued) We will show how Algorithm 2 computes $QUECA(P(u, v))$. The result should be equivalent to $\mathrm{T}_2(P(u, v))$, with 2 a finiteness point. To do so, we will show the state of variables $QUECA$ and $DTQU$ at the beginning of every iteration of the while loop in Algorithm 2.

1st iteration
$QUECA(P(u, v)) = \varnothing$
$DTQU = \varnothing : P(u, v) \bullet (R(u, v)) \wedge (\neg P(u, z) \vee v = z)$

2nd iteration
$QUECA(P(u, v)) = \varnothing$
$DTQU = R(u, v) : P(u, v) \wedge R(u, v) \bullet (\neg P(u, z) \vee v = z) \wedge (P(u, v))$

3^{rd} *iteration* $QUECA(P(u,v)) = \varnothing$
$DTQU = [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z)\bullet$
$(P(u,v)) \wedge (\neg R(u,z))] \vee$
$[(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))]$

4^{th} *iteration* $QUECA(P(u,v)) = \varnothing$
$DTQU = [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z)\bullet$
$(\neg R(u,z))] \vee$
$[(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))]$

5^{th} *iteration* $QUECA(P(u,v)) = \varnothing$
$DQU = [\neg R(u,z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)\bullet$
$(\neg P(u,z))] \vee$
$[(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))]$

6^{th} *iteration* $QUECA(P(u,v)) = \varnothing$
$DTQU = [\neg R(u,z) : P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)\bullet] \vee$
$[(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))]$

7^{th} *iteration* $QUECA(P(u,v)) = [P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)]$
$DTQU = [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z \bullet (P(u,v))]$

8^{th} *iteration* $QUECA(P(u,v)) = [P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)]$
$DTQU = [(\neg P(u,z) \vee v = z) : P(u,v) \wedge R(u,v) \wedge v = z\bullet]$

9^{th} *iteration* $QUECA(P(u,v)) = \forall z\,[[P(u,v) \wedge R(u,v) \wedge \neg P(u,z) \wedge \neg R(u,z)] \vee$
$[P(u,v) \wedge R(u,v) \wedge v = z]].$

By rearranging the result by hand, we obtain

$$\begin{aligned} QUECA(P(u,v)) &= P(u,v) \wedge R(u,v) \wedge \forall z\,[(\neg P(u,z) \wedge \neg R(u,z)) \vee v = z] \\ &= P(u,v) \wedge R(u,v) \wedge \forall z\,[(\neg P(u,z) \vee v = z) \wedge (\neg R(u,z) \vee v = z)]. \end{aligned}$$

We can see how the constraints get spread towards the related literals, in this case R, where we can see how the functional dependency of the second argument of P has generated a functional dependency for the second argument of R (due to the nature of IC). This example was shown to be syntactically non terminating for T_ω (see Example 4), but is now properly handled by *QUECA* in the sense that the finiteness point is now detected when the iteration necessarily stops due to the formulas and the syntactic conditions involved, which do not allow the process to continue. □

In the previous example we can also see how the $\bullet$ symbol works as a separator between the residues that have been included in the final query and those that are to be resolved. It graphically shows when a *TQU* is ready to be included in *QUECA*, this is when the $\bullet$ reaches the end of R, or, put in other words, when no residues are left to be resolved.

4 Properties of the *QUECA* Operator

From the point of view of database practice, we can make the natural assumption that the set of ICs is finite, and each IC in it has no more than a fixed number k of literals. Actually, if we concentrate on binary integrity constraints (see Definition 2), each IC contributes with at most two database literals. Usually, in this case, if there is one built-in per IC, we will have $k \leq 3$. Since the built-ins do not generate any residues, they are considered only at the first step. Binary ICs are also range-restricted. Since the set IC is finite, termination is guaranteed for algorithm *Res*.

The intuitive reason why *QUECA* has good properties for binary ICs is that they do not introduce disjunctive residues (except for built-ins). Having disjunctive residues would generate expansion trees associated to *QUECA*'s execution, whose branches cannot be interpreted as a conjunction of literals anymore. Instead, having BICs ensures that the tree can be represented as a disjunction of conjunctions of literals. In each conjunctive branch it is simple to check if a new residue contributes with new information or not. Disjunctions are much more complex to process in this regard. From a semantic perspective we can also see that disjunction is problematic for this query rewriting methodology (c.f. Example 16).

Theorem 1
The worst-case runtime complexity of *Res*, i.e. of Algorithm 1 that computes residues for literals wrt a set of cardinality n of universal ICs in denial form, is $O(n^2)$. □

For Algorithm 2 we have the following:

Theorem 2 (*Termination*)
For a finite set *IC* of binary ICs, Algorithm 2, which computes $QUECA(l)$ for literals l wrt *IC*, terminates in a finite number of steps. □

This termination property holds because by restricting execution to BICs only, residues contain one literal name at most, what, in the worst case, generates an infinite sequence of single literals. The potential infiniteness of this sequence is then limited by the condition in line 12 of Algorithm 2. These conditions ensure that at a given point, pending residues add no new information to the resulting query, thus being discarded.

Theorem 3
The worst-case runtime complexity of Algorithm 2, which computes $QUECA(l)$ for a literal l wrt a set of cardinality n of binary ICs, is $O(nk^{8n})$, where k is the maximum number of literals is an IC. □

Despite this complexity result, we will see in Section 6 that the process of computing residues and *QUECA* for different literals corresponds to a precomputation that can be kept stored, so it should not affect the performance from a user's point of view or data complexity. That is, once the *QUECA* queries have been precomputed for a given and fixed set of ICs (associated to some database schema or instance), the user can pose the *QUECAs* queries directly to the database, regardless of its size or whether the database contents have changed or not.

We can use Theorem 2 to extend the sufficient termination condition presented in (Arenas *et al.*, 1999) for the operator T, where semantic termination was only ensured for uniform binary integrity constraints. This is possible because we can put in correspondence the execution of the *QUECA* algorithm with the iterative application of operator T until the point where *QUECA* stops (see Examples 4 and 13). At that stage, a point of semantic termination for T is obtained. All the details of this correspondence are given in Appendix A. More precisely, we obtain

Theorem 4 (*Termination of* T)
For the class of binary integrity constraints and a literal query Q, $\mathrm{T}(Q)$ has a finite point of semantic termination, i.e. there is an $n \in \mathbb{N}$, such that $\mathrm{T}_n(Q) \equiv \mathrm{T}_m(Q)$ for all $m \geq n$. □

As a consequence of this correspondence between T and *QUECA*'s executions, we have obtained that T has a finite point of semantic termination for all BICS, and not only for uniform BICs as stated in (Arenas *et al.*, 1999). The main difference between operator T and *QUECA* for the class of BICs is that *QUECA syntactically* stops, but for some BICs, T could keep generating infinitely many syntactically different -although semantically equivalent- formulas.

Examining the difference between T and *QUECA* in more detail, we can see that T performs split operations and add residues to the pending list (for the whole set of residues) whenever *at least one* of the residues adds new information to the resulting query, whereas *QUECA* does this on a per-residue basis. This eliminates residues one by one, thus obtaining a much shorter and simpler query (c.f. $\mathrm{T}_3(P(u, v))$ in Example 4 and $QUECA(P(u, v))$ in Example 13).

We can also take advantage of the correspondence between the executions of T and *QUECA*'s, and the soundness and completeness results for T for obtaining similar results for *QUECA*.

Theorem 5 (*Soundness*)
Let r be a database instance, IC a set of binary integrity constraints and $Q(\bar{x})$ a literal query, and $\bar{t}$ a ground atom. If $r \vDash QUECA(Q)(\bar{t})$, then $\bar{t}$ is a consistent answer to Q in r, that is, $r \vDash_c Q(\bar{t})$. □

Theorem 6 (*Completeness*)
Let r be a database instance and IC a set of generic binary integrity constraints, then for every ground literal $l(\bar{t})$, if $r \vDash_c l(\bar{t})$, then $r \vDash QUECA(l)(\bar{t})$. □

The genericity condition in the completeness theorem is necessary. Consider, for example, the non generic IC $\forall x(x = a \rightarrow P(x))$, which states that the tuple a must be in table P. Every repair will contain the tuple $P(a)$, in consequence a is a consistent answer to the query $P(x)$. However this query has an empty residue; and the rewritten query is the same as the original query. It is clear that, unless a is already in table P in the original database, it will be impossible to obtain the consistent answer a by posing the query $P(x)$ to the database.

5 Queries

So far we have considered only queries that are literals. The inputs for both *Res* and *QUECA* are constants-free literals of the form $P(\bar{x})$ or $\neg P(\bar{x})$, where P represents a database table name. However, we will see now that it is possible to extend *QUECA* to queries that are completely or partially ground instantiated conjunctions of database literals and built-ins.

When conjunctions are involved, *QUECA* distributes over the conjunction. That is:

$$QUECA(l_1 \wedge \ldots \wedge l_n) \equiv QUECA(l_1) \wedge \ldots \wedge QUECA(l_n),$$

where the l_i are any literals. In case a given l_i involves a built-in predicate, *QUECA* has no effect on it, e.g. $QUECA(x \geq 7) \equiv x \geq 7$. This makes it possible to apply *QUECA* to (partially) ground queries as well. For example, $QUECA(P(x_1, \ldots, c, \ldots, x_n)) = QUECA($ $P(x_1, \ldots, x_i, \ldots, x_n) \wedge x_i = c) = QUECA(P(x_1, \ldots, x_i, \ldots, x_n)) \wedge x_i = c$.

It is easy to see that a ground query Q will generate a $QUECA(Q)$ that is a sentence, i.e. it does not have any free variables, and then, when posed against the original database instance, the answer will be *true*, i.e. the original query is true in every repair, or *false*, i.e. the original query is not true in every repair.[4] However, if the original query Q has free variables, in its *QUECA* version exactly the same free variables will appear[5]; in consequence, a (possibly empty) set of ground tuples will be returned for $QUECA(Q)$, namely those tuples that are consistent answers to Q.

[4] Notice the non symmetry between answers *true* and *false*. By the way, in our implementation we use the traditional Prolog values `yes` and `no`.

[5] Except for other variables that are used to represent the constants in the original query, as indicated above, but they will be forced to take as values the constants they represent.

Example 14. Consider the set of integrity constraints (IC) presented in Example 5:

$$\Leftarrow P(u,v) \wedge \neg R(u,v),$$
$$\Leftarrow \neg P(u,v) \wedge R(u,v),$$
$$\Leftarrow P(u,v) \wedge P(u,z) \wedge y \neq z.$$

The following database instance r violates IC

P	
a	b
a	c
d	e
f	e
g	e

R	
a	b
d	e
f	e

Using the general rewritten query $QUECA(P(u,v))$ calculated in Example 13, we may pose the queries:

Query	Answer in r
$QUECA(P(x,y))$	$\{(d,e),(f,e)\}$
$QUECA(P(x,e))$	$\{d,f\}$
$QUECA(P(a,b))$	*false*
$QUECA(P(f,e))$	*true*

If desired, these answers can be checked "semantically", by considering what is simultaneously true in the two repairs of r, namely in:

r' :

P	
a	c
d	e
f	e

R	
a	c
d	e
f	e

r'' :

P	
a	b
d	e
f	e

R	
a	b
d	e
f	e

□

Our results on complexity, termination, and soundness can be easily extended to queries that are conjunctions of literals (without existential quantifiers), in particular, to queries involving joins. Other types of queries, specifically those involving disjunction and existential quantifiers, are not covered by our solution. Actually, completeness may be lost for queries that are not conjunctions of literals.

Example 15. (existential query) Consider the database instance r of Example 14 and the query $\exists x P(a,x)$, which has *true* as a consistent answer, because in every repair there exists a tuple with a as its first argument. However, it is easy to see that the naive approach to extend $QUECA$, namely by means of $QUECA(\exists x P(a,x)) := \exists x QUECA(P(a,x))$, produces a query that is logically equivalent to $\exists x(P(u,x) \wedge R(u,x) \wedge \forall z\,[(\neg P(u,z) \vee x = z) \wedge (\neg R(u,z) \vee x = z)])$, and for this query it holds $r \not\models QUECA(\exists x P(a,x))$. The consistent answer *true* is not captured by $QUECA$. □

Example 16. (disjunctive query) Consider the query $P(a,b) \vee P(a,c)$. The database instance r of Example 14 has *true* as a consistent answer because in every repair it succeeds. However, the query $QUECA(P(a,b)) \vee QUECA(P(a,c))$ returns *false*. This shows that $QUECA$ of a disjunction is not logically equivalent to the disjunction of the $QUECA$s.

This example also illustrates why the restriction to binary constraints is relevant: the residues will contain at most one database literal, i.e. we will never have a residue containing disjunctions of database literals. Having them, might compromise the good properties $QUECA$ has (for BICs). □

In (Arenas *et al.*, 2003b; Barcelo *et al.*, 2003) methodologies for retrieving consistent

answers to arbitrary first-order queries have been proposed. They are based on disjunctive logic programs with stable model semantics. Basically the logic program specifes the database repairs in the sense that they are in a one to one correspondence with the stable models of the program. However, for the kind of queries we have considered here, the first-order query rewriting approach is more convenient, because the rewritten queries can be expressed as simple SQL queries, whose evaluation time is for sure polynomial in data complexity.

A natural question is whether there is an alternative methodology for rewriting more complex queries, e.g. involving disjunctions or existential quantifiers (projections), into first-order queries to be evaluated against the original database. Recent results show that this is unlikely or impossible. There are existentially quantified conjunctive queries (Abiteboul *et al.*, 1995) for which the CQA problem wrt to functional dependencies is coNP-complete or even Π_2^P-complete (in data complexity) (Chomicki and Marcinkowski, 2003a; Cali *et al.*, 2003). There are also existentially quantified conjunctive queries that are tractable wrt to CQA (wrt functional dependencies), but are provably not rewritable into first-order queries (Fuxman and Miller, 2003).

6 Implementation

When thinking of implementing mechanisms like the ones we have described for CQA, logic programming (LP) languages become natural candidates, because our algorithms heavily rely on purely symbolic processing of logical formulas, unification, substitution for detecting subsumption, etc. Furthermore, since in order to achieve the goal of generating queries to be posed to database, LP offers a common framework and language to capture and represent data, queries, ICs, transformation rules, etc. A detailed description of the implementation is presented in Appendix B.

We chose XSB (Sagonas *et al.*, 1994)[6] for several advantageous properties and functionalities it offers to our application (for details on features and operation see (Sagonas *et al.*, 1999)):

Tabling. Dramatically improves performance by being able to store intermediate results efficiently and thus avoid redundant subcomputation. If queries are to be posed directly from the XSB interpreter, tables can be also used as a cache for precomputed queries. Those are the queries which deliver the consistent answers, as shown in Section 5.

Relational DBMS interface. XSB comes with a general ODBC interface which allows data stored in the databases to be accessed from XSB's environment as facts for the XSB program. Intially we made XSB interact with an IBM DB2 relational DBMS, but later it was quite simple to make our implementation run in interaction with other DBMSs.

Foreign language interface. XSB may be accessed from different languages, including C and Visual Basic, which enables building powerful applications which would use XSB as a subroutine processor, or to run as a complete initialization module.

Multiplatform. XSB currently supports a number of different platforms which makes it specially convenient when working from different locations. We run the implementation under Windows Millenium.

Perhaps what makes XSB a better candidate that any other LP language is its *tabling* capabilities that improve efficiency over other systems that would, for example, have to recalculate the residues every time they are needed by Algorithm 2 (see Appendix B). Without this tabling ability, whenever the user poses a new query to a database in a same

[6] The newest version, 2.6, can be found in http://xsb.sourceforge.net/

session and precomputed queries have not been stored somewhere else, its corresponding $QUECA$ should be calculated on the fly, with possibly many redundant intermediate computations. This would diminish performance notoriously. To achieve complete persistence of the computed $QUECAs$ we should use XSB's I/O facilities to write the results to a file, so they (i.e. previous computations by $QUECA$) could be used even when the database contents has been updated.

7 Conclusions

Integrity constraints capture the semantics of a database; that is, its correspondence with the outside world the database is modelling. So, a database that satisfies its set of integrity contraints contains meaningful, semantically correct data, and stays in coordination with reality. When the database does not satisfy its ICs, then there is data in it that is in conflict with the ICs. When queries are posed to such a database, only the answers that are not in conflict should be retrieved. As discussed in the introduction, cleaning the database from conflicting information, could lead to the loss of useful information. As Example 2 shows, getting rid of the two tuples in conflict, $(a, b), (a, c)$, in table *Provider*, would lead to loosing the correct piece of information that a is a provider.

Our approach, namely consistent query answering, is more flexible and realistic. There are many situations where data cleaning with the purpose of restoring consistency is not an alternative. In our framework, each -non privileged- user of a central DBMS could easily impose, at query time, his/her own user ICs on the available data.

The semantics of consistent query answering was introduced in (Arenas *et al.*, 1999). It is based on the auxiliary notion of minimal repair, i.e. of a new but consistent database instance that differs from the original one by a minimal set of tuples under set inclusion. However, when computing consistent query answers, the idea is to avoid computing the repairs, for obvious reasons. Ideally, only the available and inconsistent original database should be used. The query rewriting approach, first introduced in (Arenas *et al.*, 1999) and expanded and implemented here, fully captures this idea, at least for certain useful and interesting classes of queries and ICs. Most of the ICs found in database praxis can be handled by our algorithms and implementation, except for referential ICs (Abiteboul *et al.*, 1995), usually involved in foreign key constraints.

We established our methodology to be sound, complete and terminating for an interesting set of ICs, the binary BICs. The complexity of the algorithm was also analyzed. Finally, the procedures were implemented in XSB, using standard logic programming techniques, but taking advantage of powerful XSB's evaluation strategies. XSB was coupled to an ODBC compliant database, in this case, IBM DB2, to build an interactive querying system. We think the whole implementation could easily and fruitfully become a part of an enhanced and standard DBMS technology.

As mentioned in Section 5, there are fundamental limitations for extending the first-order query rewriting approach to more complex queries and existential ICs, like referential ICs. Those cases can be handled with the logic programming based approach (Arenas *et al.*, 2003b; Greco *et al.*, 2003; Barcelo and Bertossi, 2003; Barcelo *et al.*, 2003). An annotated logic based approach to consistent query answering is presented in (**?**), where repairs are characterized as preferred models of a theory, however algorithmic solutions are not fully developed.

However, even in the case of more complex queries and ICs, those not covered by the methodology presented here, our approach and implementation can be used to compute *the consistent core* of the database, i.e. the restrictions of the original tables to the intersection of all repairs. This consistent core can be used to compute answers to more involved queries, by splitting computation into the core and the "inconsistent part". This idea has already

been explored in (Arenas *et al.*, 2003a) in the context of aggregate queries, and also in the logic programming based approach to CQA (Bertossi and Bravo, 200?).

Our notion of consistent query answer is based on the notion of database repair introduced in (Arenas *et al.*, 1999). Repairs have a minimal distance to the original database wrt set inclusion of whole tuples. However, other notions of repair could be considered for certain scenarios and applications, but there have been only a few first attempts to study them, e.g. repairs based on minimal *number* of tuples (Arenas *et al.*, 2003b; Franconi *et al.*, 2001), repairs based on corrections of attribute values (Wijsen, 2003; Franconi *et al.*, 2001), preference based approaches (Greco *et al.*, 2003; Chomicki and Marcinkowski, 2003a; Franconi *et al.*, 2001). In a different direction, independently from the notion of repair adopted, considering as consistent answers those that are true in, e.g. the *majority* of the repairs, could be an interesting research direction. Implementation issues are still to be addressed in most of those cases.

Connections of our line of research with other approaches to inconsistency handling in data and knowledgebases, and with belief revision/update are discussed in (Bertossi and Chomicki, 2003).

With respect to future work, it is necessary to bring closer the query rewriting and the logic programming approaches to CQA, understanding better their differences and similarities. Both approaches are necessary as explained above, due to complexity reasons, however research on them has gone largely through separately trajectories. In (Bertossi and Bravo, 200?) the collapse of logic programs for CQA to first-order queries is being investigated. A useful implementation should be able to combine the two approaches, using the strength of each them when appropriate. Of course, the two approaches return the same answers, but they are not equally expensive in term of computational complexity and implementation.

Much research is needed around implementation issues. The only implementation of the query rewriting based approach we are aware of is the one presented in this paper. Future research in this direction includes optimization of the resulting queries, both syntactically and semantically; elimination of semantically redundant residues, i.e. related to built-in predicates; and experimentation with real, large relational databases.

Implementation issues around the logic programming based approach are reported in (Arenas *et al.*, 2003b; Greco *et al.*, 2003; Barcelo and Bertossi, 2003; Barcelo *et al.*, 2003; Eiter *et al.*, 2003). All of them give a disjunctive stable model semantics to the logic programs, and use the DLV system for experimentation (Eiter *et al.*, 2000). An implementation of CQA for non-existentially quantified conjunctive queries based on graph theoretical methods and algorithms introduced in (Chomicki and Marcinkowski, 2003a) is reported in (Chomicki *et al.*, 2003b; Chomicki *et al.*, 2004).

Acknowledgements: This research has been supported by FONDECYT (grant 1000593), NSERC (grant 250279-02), and DIPUC. L. Bertossi is a Faculty Fellow of the IBM Center for Advanced Studies, Toronto Lab. We are grateful to Marcelo Arenas for useful and illuminating conversations. We are also grateful to Baoqiu Cui for his support with XSB's ODBC interface.

References

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

Arenas, M.; Bertossi, L.; and Chomicki, J. Consistent Query Answers in Inconsistent Databases. In *Proc. ACM Symposium on Principles of Database Systems (ACM PODS'99, Philadelphia)*, ACM Press, 1999, pp. 68–79.

Arenas, M.; Bertossi, L.; Chomicki, J.; He, X., Raghavan, V.; and Spinrad, J. Scalar

Aggregation in Inconsistent Databases. *Theoretical Computer Science*, 2003, 296(3):405-434.

Arenas, M.; Bertossi, L.; and Chomicki, J. Answer Sets for Consistent Query Answers. Theory and Practice of Logic Programming, 3, 4&5, 2003, pp. 393-424.

Arenas, M.; Bertossi, L. and Kifer, M. Applications of Annotated Predicate Calculus to Querying Inconsistent Databases. In *'Computational Logic - CL2000' Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000)*. Springer Lecture Notes in Artificial Intelligence 1861, pages 926-941.

Barcelo, P.; and Bertossi, L. Logic Programs for Querying Inconsistent Databases. Proc. Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003). Springer Lecture Notes in Computer Science 2562, 2003, pp. 208–222.

Barcelo, P.; Bertossi, L.; and Bravo, L. 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.

Bertossi, L.; Arenas, M.; and Ferretti, C. SCDBR: An Automated Reasoner for Specifications of Database Updates. *Journal of Intelligent Information Systems*, 1998, 10(3): 253–280.

Bertossi, L.; and Chomicki, J. Query Answering in Inconsistent Databases. In 'Logics for Emerging Applications of Databases', J. Chomicki, G. Saake and R. van der Meyden (eds.), Springer, 2003.

Bertossi, L.; and Bravo, L. On Query Rewriting for Consistent Query Answering. In preparation.

Böhlen, M. Managing Temporal Knowledge and Data Engineering. Phd. Thesis #10802, ETH, Zürich, 1994.

Burse, J. ProQuel: Using Prolog to Implement a Deductive Database System. Technical Report. Department Informatik, ETH Zürich, 1992.

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)*, 2003, pp. 260-271.

Celle, A. Implementing Consistent Query Answers in Inconsistent Databases. M.Sc. Thesis, Depto. Ciencia de Computación, P. Universidad Católica de Chile, Santiago, Chile, 2000. Available at `http://www.scs.carleton.ca/~bertossi/papers/master.pdf`

Celle, A.; and Bertossi, L. Querying Inconsistent Databases: Algorithms and Implementation. In *'Computational Logic - CL 2000'. Stream: 6th International Conference on Rules and Objects in Databases (DOOD'2000)*, Springer Lecture Notes in Artificial Intelligence 1861, 2000, pp. 942–956.

Chakravarthy, U.S; Grant, J.; and Minker, J. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 1990, 15(2): 162–207.

Chomicki, J.; and Marcinkowski, J. Minimal-Change Integrity Maintenance Using Tuple Deletions. arXiv.org paper cs.DB/0212004.

Chomicki, J.; Marcinkowski, J.; and Staworko, S. Consistent Answers for Quantifier Free Queries. In Proc. Dagstuhl Seminar on Inconsistency Tolerance, July 2003. http://www.dagstuhl.de/03241/Proceedings/

Chomicki, J., Marcinkowski, J. and Staworko, S. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. To appear as system demonstration in Proc. Conference on Extended Database Technology (EDBT 04), Springer, 2004.

Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. Declarative Problem-Solving in DLV. In 'Logic-Based Artificial Intelligence', J. Minker (ed.), Kluwer, 2000, pp. 79–103.

Eiter, T.; Fink, M.; Greco, G.; and Lembo, D. Efficient Evaluation of Logic Programs for

Querying Data Integration. In Proc. of the 19th Int. Conference on Logic Programming (ICLP 2003), 2003.

Franconi, E.; Laureti Palma, A.; Leone, N.; Perri, S.; and Scarcello, F. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001). Springer LNCS 2250, 2001, pp. 561-578.

Fuxman, A.; and Miller, R.J. Towards Inconsistency Management in Data Integration Systems. Proceedings of the IJCAI-03 Workshop on Information Integration on the Web. On-line proceedings: http://www.isi.edu/info-agents/workshops/ijcai03/proceedings.htm.

Greco, G., Greco, S., Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. IEEE Trans. Knowledge and Data Engineering, 2003, 15(6):1389-1408.

Kanellakis, P.C. Elements of Relational Database Theory. In 'Handbook of Theoretical Computer Science', vol. B, Van Leeuwen (ed.), Elsevier/MIT Press, 1990, pp. 1073–1158.

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

McCune, W.W. OTTER 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

Nicolas, J.-M. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 1982, 18(3): 227–253.

Rao, P., Sagonas, K.F, Swift, T., Warren, D.S. and Freire, J. XSB: A System for Efficiently Computing WFS. In Proc. 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997). Springer Lecture Notes in Computer Science 1265, 1997, pp. 431-441.

Reiter, R. Towards a Logical Reconstruction of Relational Database Theory. In 'On Conceptual Modelling', Brodie, M.; Mylopoulos, J.; and Joachim Schmidt (eds.), Springer-Verlag, 1984.

Sagonas, K.F.; Swift, T.K.; and Warren, D.S. XSB as an Efficient Deductive Database Engine. In Proceedings SIGMOD'94, ACM Press, 1994, pp. 442-453.

Sagonas, K.F.; Swift, T.; Warren, D.S.; Freire, J.; and Rao, P. The XSB System Version 2.0, Volume 2: Libraries and Interfaces, 1999.

Topor, R.W.; and Sonenberg, E.A. On Domain Independent Databases. In 'Foundations of Deductive Databases and Logic Programming', Minker, J. (ed.), Morgan Kaufmann Publishers, 1988, pp. 217–240.

Ullman, J.D. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

Van Gelder, A.; and Topor, R.W. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 1991, 16(2): 235–278.

Vardi, M.Y. The Complexity of Relational Query Languages. In Proc. STOC, 1982, pp. 137–146.

Wijsen, J. Condensed Representation of Database Repairs for Consistent Query Answering. In Database Theory - ICDT 2003, Springer LNCS 2572 ,2003, pp. 378-393.

A Proof of Results

Proof of Theorem 1: We determine complexity functions, F_i, for different parts of the Algorithm 1. The numbers on the left represent the corresponding line numbers in it.

1–1: $F_1(n) = (c_1 - c_3)n + c_2n^2$.

Read the ICs and insert them into a list L_{IC} is c_1n. Next, we must build a list, L_P, of distinct literal names appearing in the ICs. Since we accept only BICs, the maximum number of literal names per constraint is 2, so in the worst case (when they are all different) we have $2n$ literal names. The number of comparisons performed is $\sum_{i=1}^{2n}(i-1) = c_2n^2 - c_3n$. So, $F_1(n) = c_1n + c_2n^2 - c_3n = (c_1 - c_3)n + c_2n^2$.

2–13: $F_2(n) = c_4kn^2$.

By considering that $|L_P|$ is bounded by $2n$, we have that the loop executed between lines 5–10 is repeated a maximum of $2n^2$ times. The maximum number of comparisons performed in line 5 is k, where k is the maximum number of literals in an integrity constraint, so $F_3(n) = c_4kn^2$.

14–21: $F_3(n) = c_5(k-1)^2n^2 - c_6(k-1)^2n$.

If all literal names are different, we have one residue for each, and if they are all equal we have $2n$ residues for that single one. So, in either case, the redundancy elimination process must be executed $2n$ times. This process involves checking that every literal belongs to a single residue already in the final list for a given substitution and if it succeeds, the reversal must be performed. Now, the worst case would be that no two residues are redundant and the checking process fails in the last step, that is, the last term of the reversal process, so all the possible checks must be made. So, for a residue with $(k-1)$ literals we would have to do $(k-1)^2$ checks to determine if it is redundant or not with another residue with $(k-1)$ literals. This can be formulated as $\sum_{i=1}^{2n}(c(i-1)(k-1)^2)$. So $F_3(n) = c_5(k-1)^2n^2 - c_6(k-1)^2n$.

The resulting runtime complexity of Algorithm 1 is $F(n) = F_1(n) + F_2(n) + F_3(n) = (c_1 - c_3)n + c_2n^2 + c_4kn^2 + c_5(k-1)^2n^2 - c_6(k-1)^2n = O(n^2)$. □

Before analyzing the runtime complexity of Algorithm 2, we must define a notion related to the critical decision point in the algorithm's execution. We will call it a *failure*.

Definition 7 (*Failure*)
A *failure* in Algorithm 2 occurs when the condition in line 9 fails and the one in line 20 succeeds at least once. Otherwise it is called a *success*. □

A failure increases the number of literals in $DTQU$ by a maximum of $k-1$, being k the number of literals per integrity constraint. On the other hand, a success decreases the number of literals in $DTQU$ by one. The maximum number of failures that may occur must be bounded in order to be able to calculate the runtime complexity of Algorithm 2.

Lemma 1
Restricting IC to BICs with a maximum of k literals each, and being $|IC| = n$, the maximum number of failures in Algorithm 2 is upper bounded by $\frac{k^{4n+1}-1}{k-1}$.

Proof: The execution of Algorithm 2 can be represented as a tree in which l is the root and the literals from the first residue are its sons, the literals from the second, its grandsons, and so forth. Following this interpretation, what delivers children is a failure, so the number of inner nodes of the tree indicates how many failures have occurred.

We know that by restricting IC to BICs we have a maximum of $2n$ different literal names each having a maximum of 1 literal name per residue. Because of the condition in line 24 and the fact that ICs are range restricted denials (6), we can have a maximum of $2n$ predicates or negated predicates in any path from root to leaf, the rest are built-ins. These built-ins are either part of the path or will be when the pending residues are resolved. Only literal names generate residues, so there is a maximum of $(2n)^2$ residues that

will supply built-ins ($2n$ residues per each of the $2n$ literal names). However, because of the condition in line 13, once the first $2n$ built-ins have been added, all the residues are stored in D, so no more failures will occur. Thus we have that the maximum depth of the tree would then be $4n$. And consequently the number of failures (inner nodes) is upper bounded by $\sum_{j=0}^{4n} k^j = \frac{k^{4n+1}-1}{k-1}$. □

Proof of Theorem 2: It follows immediately from Lemma 1. □

Having an upper bound for failures, it is possible to calculate the runtime complexity of Algorithm 2.

Proof of Theorem 3: Before starting this calculation we introduce a new useful quantity, r, the maximum number of residues associated to any literal. It is easy to see that $0 \leqslant r \leqslant 2 \cdot n$. Again, the numbers on the left represent the corresponding line numbers in the algorithm.

1–2: $F_1(n) = c_1 + c_2 r$.

The runtime complexity of line 2 is clearly linear wrt the number of residues associated to the literal.

Now, let f denote the maximum number of *failures* of the algorithm until completion.

3–31: $F_2^{(f)}(n) = c_3(k-2)(r-1)f^2 + c_3 k(r-1)f + r$.

Clearly the availability of literals in $DTQU$ determines how many executions of the while loop will take place. Furthermore, the number of executions is really bounded by the number of pending residues at the right of $\bullet$ in every literal appearing in $DTQU$s. The upper bound of this number will be given by the maximum number of literals in $DTQU$s times the maximum number of residues one of them has at a given moment. So, once the algorithm is initialized, a first check takes place in line 5. We will assume the worst case, which is that this condition is never met and so, the algorithm never stops. Next, if the residue checks succeed in line 9, no new residues are introduced, and the number of residues of the first term of $DTQU$s is reduced by one, so whatever the upper bound we picked it still remains as such. On the other hand, if a failure occurs, the upper bound must be modified to include at most $(k-2)$ new literals, and only one of them adds at most $r-1$ new residues (because we are dealing with BICs the rest of the residue are only built-ins which do not add residues, so now they have one residue less).

Now, we are ready to calculate the upper bounds. At the beginning, we have $DTQU = \varnothing : l \bullet residues(l)$, that is, there is only one literal in $DTQU$ ($|DTQU| = 1$) and a maximum of r residues yet to be resolved in that literal. In consequence, $F_2^{(0)}(n) = 1 \cdot r$.

Next, a *failure* occurred, so now the maximum number of literals is $1+(k-2) = (k-1)$, with the first literal having a maximum of $r+(r-1) = 2r-1$ residues. This procedure can be iterated to obtain

$$\begin{aligned} F_2^{(1)}(n) =& (2r-1) + (k-2)(r-1). \\ F_2^{(2)}(n) =& 1\Delta(2r-1+r-1) + (k-2)(2r-2) + (k-2)(r-1). \\ =& (3r-2) + (k-2)(2r-2) + (k-2)(r-1). \end{aligned}$$

$$\begin{aligned}
F_2^{(3)}(n) =& 1\Delta(3r-2+r-1)+(k-2)(3r-3)+(k-2)(2r-2)+\\
& (k-2)(r-1).\\
=& (4r-3)+(k-2)(3r-3)+(k-2)(2r-2)+(k-2)(r-1).\\
F_2^{(4)}(n) =& ...
\end{aligned}$$

In this way we obtain:

$$\begin{aligned}
F_2^{(f)}(n) &= (f+1)r-f+(k-2)(r-1)\sum_{i=1}^{f} i\\
&= O((k-2)(r-1)f^2+k(r-1)f+r).
\end{aligned}$$

The resulting runtime complexity of Algorithm 2 is $F(n) = F_1(n) + F_2^{(f)}(n) = O((k-2)(r-1)f^2 + k(r-1)f + r)$, where f ($\geqslant 0$) represents the maximum number of *failures* of the algorithm until completion, k is the maximum number of literals per integrity constraint and $r \leqslant k \cdot n$ is the maximum number of residues a literal may have for a given set of n ICs. Since r is upper bounded by $k \cdot n$ and k is upper bounded by a constant, we obtain that the runtime complexity of $QUECA(l)$ for one literal is $F(n) = O(nf^2)$. But now, from Lemma 1, we know that the number of *failures* is bounded by $O(k^{4n})$; then the runtime complexity for $QUECA(l)$ is $F(n) = O(nk^{8n})$. □

In order to establish soundness for the operator $QUECA$, we must first prove that the notion of information redundancy of a residue introduced in Definition 6 is semantically correct, in the sense that the residue is logically implied by the already available information.

Lemma 2
Given a $TQU \quad D:E \bullet R$, let $\phi = l_1 \vee \cdots \vee l_n \in R$ be a residue. If $\phi \,\widetilde{\in}\, D:E$, then either $E \vDash \phi$ or $\bigvee_{\{i|D_i \in D\}} E_i \vDash \phi$. □

Proof: For a substitution θ as in Definition 6 we have two cases, namely: (a) when for all i, $l_i\theta \in E$; and (b) when $\phi\theta \in D$.

(a) Assume that for all i, $l_i\theta \in E$. Since θ substitutes variables occurring nowhere else but in ϕ for variables which are universally quantified in E, and because we know that $l_1 \wedge l_2 \vDash l_1 \vee l_2$, we have that $E \vDash \phi$.

(b) Assume that $\phi\theta \in D$. We know that D contains the last residue that generated a split operation plus all those residues φ, if any, such that $\varphi \;{}_P\widetilde{\in}_\theta\; D:E$ for the case that no literal is added to its E (line 13). We also know that all the elements from DTQUs whose D share elements have been involved in a split operation, so they have a common stem e and are just differentiated by the last portion of E. In this way $\bigvee E_i$, over all the i's such that $D_i \in D$, can be rewritten as $e \wedge D$. Since θ substitutes variables occurring nowhere else but in ϕ for variables which are universally quantified in D, we have that $\phi\theta \in D$ implies $D \vDash \phi$. In consequence, we have $e \wedge D \vDash \phi$, which implies that $\bigvee_{\{i|D_i \in D\}} E_i \vDash \phi$. □

Now it is possible to prove the soundness of Algorithm 2. In order to do this, the execution of $QUECA$ will be mapped to that of T with the purpose of taking advantage of T's soundness and completeness results presented in (Arenas *et al.*, 1999). First, residues can be classified into generations according to the following inductive definition.

Definition 8
1. Given a literal query Q, *residues*(Q) are first generation residues.
2. Residues of n^{th} generation residues are $(n+1)^{th}$ generation residues. □

Lemma 3

Restricting *IC* to BICs, for a literal query Q the execution of Algorithm 2 (stops and) delivers a first-order formula $QUECA(Q)$, such that for some $n \in \mathbb{N}$, $QUECA(Q)$ and $\mathrm{T}_n(Q)$ are logically equivalent ($\equiv$), and n is a finiteness point of semantic termination of operator T for Q.

Proof: The execution of Algorithm 2 follows a depth first search strategy in the sense that it systematically selects the first element from DTQUs to resolve its first pending residue. However, this is not essential for sound execution because *all* branches of the tree *must* be followed and completely evaluated. In this way the elements of DTQUs can be selected in *any* order without compromising soundness.

Having said this, and taking into account the residue generations presented in Definition 8, the elements of DTQUs can be selected in such a way that all the elements having first generation residues are selected in first place. Then, when no first generation residues are left in any element belonging to DTQUs, elements having second generation residues are selected. This selection schema continues until completing the execution. As auxiliary notation, when all k generation residues have been resolved, the resulting (partial) rewriting will be denoted by $QUECA^k(\cdot)$, being $QUECA$ formed by the disjunction of all the E's present at that moment in DTQUs. In this way we can map $QUECA$'s execution to that of operator T, presented in (Arenas *et al.*, 1999), in the following way:

$$\begin{aligned}
&\mathrm{T}_0(Q) = Q \equiv QUECA^0(Q),\\
&\mathrm{T}_1(Q) = Q \wedge \mathit{residues}(Q) \equiv QUECA^1(Q),\\
&\cdots\\
&\mathrm{T}_n(Q) = \ldots \equiv QUECA^n(Q) = QUECA(Q),
\end{aligned}$$

for some natural number n. Since only BICs are being considered, stopping is guaranteed at some finite step n (see Theorem 2).

Although not all residues are added to $QUECA$, Lemma 2 states that the information represented by residues that were discarded before reaching step n is already semantically included in $QUECA$; thus it is irrelevant. This information discarding is what makes the algorithm stop. However, this stopping condition (i.e. lines 9–18, 20 and 24–26) can be stripped off the algorithm, and the resulting formula at step n will be logically equivalent to the original one; it will also be equivalent to T_n. Even more, the execution of $QUECA$ could then be taken one step further, to when all $n+1$ generation residues are resolved. This step is not reached originally (with the stopping conditions in place) because all the information added in it is redundant, that is $QUECA(Q)^n \equiv QUECA(Q)^{n+1} \equiv QUECA(Q)$ (see Lemma 2).

Consequently, by transitivity and the invariant described earlier, it is proved that $\mathrm{T}_n(Q) \equiv QUECA(Q)$ and $\mathrm{T}_n(Q) \equiv \mathrm{T}_{n+1}(Q)$, so n is the finiteness point as defined in (Arenas *et al.*, 1999). □

Proof of Theorem 4: Immediately obtained from the equivalence established in the proof of Lemma 3 and the termination of $QUECA$ established in Theorem 2. □

Proof of Theorem 5: From Lemma 3 and Theorem 2 in (Arenas *et al.*, 1999). □

Proof of Theorem 6: From Lemma 3 and Theorem 4 in (Arenas *et al.*, 1999). □

Table B 1. *Built-in operators allowed in Integrity Constraints*

System's Built-in	Represents
~F	$\neg F$
T_1 == T_2	$T_1 = T_2$
~(T_1 == T_2)	$T_1 \neq T_2$
T_1 < T_2	$T_1 < T_2$
T_1 =< T_2	$T_1 \leq T_2$
T_1 > T_2	$T_1 > T_2$
T_1 >= T_2	$T_1 \geq T_2$

B Implementation Details

B.1 Program Overview

As could be noted in Section 3, the only input for the algorithms presented in this article is the set of integrity constraints, *IC*.[7] Therefore, representing *IC* in an appropriate manner is important. Next, when posing a query to a database, we need to use the ODBC interface of XSB to access the database directly. For this the database schema has to be properly defined. These two issues are described next.

Integrity Constraints: Integrity constraints are to be defined in a file named `ics` in conformity with the syntax of (6), that is, each IC has the representation:

```
<- [ ... denials ... ].                                    (B 1)
```

Each constraint must go in a separate line. Inside the brackets, as in traditional Prolog lists, literals must be separated by commas which represent conjunctions. They have to be written according to the standard Prolog convention of capitalizing variables and leaving object constants in lower case. The built-in operators allowed in an IC and their syntax are shown in Table B 1.

Example 17. The ICs in Example 5 would be represented as follows:

```
<- [p(U,V),~r(U,V)].
<- [~p(U,V),r(U,V)].
<- [p(U,V),p(U,Z),~(V==Z)].
```

Other examples of ICs:

```
<- [m(X,Y),X=<10].
<- [q(X),X==apple].
<- [q(X),X==orange].
<- [t(X),s(Y),X>Y].
```

□

[7] The literals for which relevant residues should be generated can be read directly from *IC*. Once this is done, Algorithms 1 and 2 can compute the residues and the results of QUECA for all the general, relevant literals in the ICs, resp.

This file *must* reside in the same directory as XSB's binary executable for the system to find it.

Data Objects: By taking advantage of XSB's RDBMS interface, the program does not need to transform the data into a suitable format for processing. Consequently, having an adequate ODBC driver for the database in which the data is stored, allows us to access it directly from the program.[8] In order to access the data stored in the database effectively, we must take care to define the tables with lower case names and include a *dummy* table, named `dummy`, containing just one column (c1) and one element (X).

B.2 Using the Program

In this section we briefly describe how to use the system. We will begin by defining the syntax of the queries to be posed to the system. Next, we will show how to retrieve *consistent* information from the database. Some other useful predicates will be presented before analyzing some particular issues regarding a system module of XSB and the 'allowedness' of queries.

Queries: The queries supported by the system are the non quantified conjunctions of literals described in Section 5. Usual Prolog conventions on variables and constants apply. For example, some admissible queries are `prod(X,Y).` and `client(alex,Z,santiago).`, where the tables `prod` and `client` are defined in the database.

The usual restrictions on queries apply. In order to ensure *domain independence* for them (Ullman, 1988), decidable, proper syntactic subclasses were introduced in, e.g. (Nicolas, 1982; Topor and Sonenberg, 1988; VanGelder and Topor, 1991). For more details see (Abiteboul *et al.*, 1995).

To translate FOL formulas into SQL queries, avoiding to code a translator from scratch, we took advantage of a module coded for SCDBR (Bertossi *et al.*, 1998), an automated reasoner on specifications of database updates. The specific condition we imposed on queries to ensure domain independence is that of being *allowed* as defined in (Böhlen, 1994; Burse, 1992), that is slightly different from the original one in (Topor and Sonenberg, 1988). The main advantage of this notion is that recognizing an allowed formula and then translating it into relational algebra is fairly simple.

To deal with conjunctive queries, the user must pose them manually, based on the results written to the file `results` via the predicate `list_all/2` (see the item on retrieving consistent answers below).

Program Initialization: To initialize the system, the user must do the following:

1. The XSB interpreter must be started and `| ?- [main].` must be executed to consult the main module of the program.
2. `| ?- install.` must be executed. This instruction compiles the rest of the modules.
3. `| ?- init.` must be executed. This predicate consults the rest of the modules and then connects to the specified database. This is where the whole *QUECA* calculation routine is executed.
   Once the initialization procedure is completed, the *QUECAs* for the relevant literals in the ICs and their equivalent SQL queries are stored both in XSB's tables, and also in the file `results` for further reference.

[8] We used XSB version 2.1. For this interface to work properly, a patch -kindly provided by the members of XSB's development team- was necessary.

Table B 2. *Predicates and their meaning*

System's Predicate	Represents
$no(F)$	$\neg F$
$and(F_1, F_2)$	$F_1 \wedge F_2$
$or(F_1, F_2)$	$F_1 \vee F_2$
$equal(T_1, T_2)$	$T_1 = T_2$
$T_1 =< T_2$	$T_1 \leq T_2$
$T_1 < T_2$	$T_1 < T_2$
$T_1 >= T_2$	$T_1 \geq T_2$
$T_1 > T_2$	$T_1 > T_2$
$all(X, F)$	$(\forall x)\ F$

Retrieving Consistent Information: Having successfully completed the initialization procedure, the user may execute:

- `| ?- query(Q,R).` Given a query `Q`, it returns in `R` a *consistent* tuple. It consults the database directly, using the corresponding *QUECA* for `Q`. For example, these are queries: `query(prod(X,Y),R)`, `query([prod(X,Y),Y > 7],R)`, `query([prod(X,Y), trans(X,Y)],R)`, `query([prod(X,Y),~trans(Y,Z)],R)`.
  Other consistent answers can be obtained by backtracking. Notice that in spite of the fact that `query(Q,R)` generates an underlying SQL query, XSB returns one tuple at a time. An alternative for obtaining *all* the expected answers consists in getting them as a list `L` of tuples by using predicate `findall(G,query(Q,R),L)`, which is a built-in in XSB. Here, `G` is a list of variables, and the variables in query `Q` that do not appear in `G` are treated as existential variables. Since we are computing consistent answers to non existential queries, in `findall(G,query(Q,R),L)`, `G` has to be specified as the list of all variables in `Q` (that coincides with the list of free variables in $Queca(Q)$). A query could be: `findall(G,query([P(U,V),~R(U,V)]),Result),L)`.

Other Useful Predicates: Other predicates available to the user after initialization are:

- `| ?- residues(Q,R).` Given a literal query `Q`, its residues are returned in `R` as a list in CNF (i.e. a inner list represents a clause and the list of lists is a conjunction of them). These residues are used to compute *QUECA* for literal queries.
- `| ?- queca(Q,R).` Given a query `Q`, its calculated $QUECA(Q)$ is returned in `R` as a first-order formula in prefix form, that uses the notation in Table B 2 (see Example 18 below).
- `| ?- qca2sql(Q,R).` In case the formula calculated by `queca(Q,R)` is *allowed*, it returns in `R` the SQL query equivalent to the $QUECA(Q)$ already computed. Otherwise, it returns the string *Formula not allowed* (see Example 19).

The predicates above and others are "tabled" in XSB, and this has been declared as a part of the system's implementation using statements like

```
:- table ic_pred_list/3.  :- table residues/2.  :- table queca/2.
:- table facts/2. :- table qca2sql/2.
```

This has the effect of caching intermediate results, so that redundant computations, e.g. if the same query is posed twice, are avoided and the file `results` does not need to be

consulted. The computation of the *QUECA* queries and their SQL versions is accelerated, because the residues for the different predicates are kept cached.

Ending the Application: To terminate the application the following predicates are included:

- `| ?- end.` Disconnects from the database.
- `| ?- halt.` Exits XSB.

Example 18. Assuming we have modified the file `ics` to contain the integrity constraints defined in Example 17, we would obtain the following results:

```
| ?- [main].
yes
| ?- install.
yes
| ?- init.
yes

| ?- queca(p(X,Y),Q).
Q =
and(p(id1,id2),all(u1,and(r(id1,id2),or(and(no(p(id1,u1)),no(r(id1,u1))),
    equal(id2,u1)))))
yes

| ?- qca2sql(p(X,Y),Q).
Q =  SELECT a0.c1,a0.c2 FROM  "p"  a0
WHERE NOT EXISTS
    (SELECT '*' FROM  "r"  a3 WHERE a0.c2<>a3.c2 AND a3.c1=a0.c1) AND
NOT EXISTS
    (SELECT '*' FROM  "p"  a2 WHERE a0.c2<>a2.c2 AND a2.c1=a0.c1) AND
NOT EXISTS
    (SELECT '*' FROM  "dummy"  WHERE NOT EXISTS
    (SELECT '*' FROM  "r"  a1 WHERE a1.c1=a0.c1 AND a1.c2=a0.c2))
yes
| ?- end.
yes
```

For an explanation of what table `dummy` represents see Section B.1.

In order to get the consistent tuples of $P(x, y)$ we would pose the following query in the XSB environment `query(p(X,Y),Q)`. Internally its *QUECA* and SQL version would be computed. □

Example 19. Consider the two non generic ICs $\forall x(x = a \rightarrow S(x))$ and $\forall x(x = a \rightarrow \neg T(x))$. Fed into the system, they become `<- [X==a,~s(X)]. <- [X==a,t(X)].`

Running the system, the literals $\neg S(x)$ and $T(x)$ get the following residues

```
PREDICATE : ~s(_h92)
     RESIDUES : [[~(_h92 == a)]]
PREDICATE : t(_h105)
     RESIDUES : [[~(_h105 == a)]]
```

that is, $\neg x = a$ for both of them. The literals $S(x)$ and $\neg T(x)$ do not have any residues.

The *QUECAS* for the literals $S(x), \neg S(x), T(x), \neg T(x)$ are

```
PREDICATE : s(_h92)
     QUERY : s(id1)
PREDICATE : ~s(_h92)
     QUERY : and(no(s(id1)),no(equal(id1,a)))
PREDICATE : t(_h105)
     QUERY : and(t(id1),no(equal(id1,a)))
PREDICATE : ~t(_h105)
     QUERY : no(t(id1))
```

that is, $S(x)$, $(\neg S(x) \wedge x \neq a)$, $(T(x) \wedge \neg x = a)$, $\neg T(x)$, resp. The translation into SQL produces:

```
PREDICATE : s(_h92)
     SQL   : SELECT a0.c1 FROM "s" a0
PREDICATE : ~s(_h92)
     SQL   : Formula not allowed
PREDICATE : t(_h105)
     SQL   : SELECT a0.c1 FROM  "t"  a0 WHERE a0.c1<>'a'
PREDICATE : ~t(_h105)
     SQL   : Formula not allowed
```

We can see that the system detects when the generated query is not a safe query, in which case it returns the "`Formula not allowed`" message. □

For a much more detailed description of the implementation and its use see (Celle, 2000).