Querying Inconsistent Databases: Algorithms and Implementation

Alexander Celle and Leopoldo Bertossi

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de Computación
Casilla 360, Santiago 22, Chile
{acelle,bertossi}@puc.cl

Abstract. In this paper, an algorithm for obtaining consistent answers to queries posed to inconsistent relational databases is presented. This is a query rewriting algorithm proven to be sound, terminating and complete for some classes of integrity constraints that extend those previously considered in [1]. Complexity issues are addressed.
The implementation of the algorithm in XSB presented here takes advantage of the functionalities of XSB, as a logic programming language with tabling facilities, and the possibility of coupling it to relational database systems.

1 Introduction

It is usually assumed that data stored in a database is consistent; and not having this consistency is considered a dangerous situation. However, it often happens that this is not the case and the database reaches an inconsistent state in the sense that the database instance does not satisfy a given set of integrity constraints *IC*. This situation may arise due to several reasons. The initial problem was due to poor design of the database schema itself or a malfunctioning application that made the system reach the inconsistent state.

Nowadays, other sources of inconsistencies have appeared. For example, in a datawarehouse context [4], inconsistencies may appear, among other reasons to integration of different data sources. In particular, in the presence of duplicate information, and to delayed update of the datawarehouse views.

Either case, having a consistent database or not, the information stored in it remains relevant to the user and is potentially useful, as long as the distinction between consistent and inconsistent data can be made, and they can be separated when answering queries.

The common solution for the problem of facing inconsistent data is to repair the database and take it back to a consistent state. However, this approach is very expensive in terms of computing power, complexity and in some cases we might lose potentially relevant data in the process. In addition, a particular user, without control on the database administration, might want to impose his/her particular, soft or hard constraints on the database (or some views). In this case, the database cannot be repaired.

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

Purchase		Client
c	e_1	c
d	e_2	
d	e_1	

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

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 we can still use the inconsistent data for analysis (purchases of customer d in Example 1).

A semantic notion of consistent answer to a query was given in [1]. In essence, a tuple answer $\bar{t}$ is a consistent answer to a query $Q(\bar{x})$ if $Q(\bar{t})$ becomes true in every repair of the inconsistent database instance r that can be obtained by a minimal set of changes on r. Of course, the idea is not to construct all possible minimal repairs and then query; this would impossible or too complex. It is necessary to search for an alternative mechanism.

In this context, an operator T_ω was presented in [1] which does not repair the database but that, given the query $Q(\bar{x})$, computes a modified query $\mathrm{T}_\omega(Q)(\bar{x})$ whose answers, when posed to the original database instance r, are consistent in the semantic sense already explained. The operator produces query rewritings that are sound, complete and terminating for interesting syntactic classes of queries and constraints [1]. However, this operator has some drawbacks: it is hard to implement due to its recursive nature and a semantic termination condition.

In this paper we address the problem of designing and implementing an alternative operator inspired by T_ω. The new operator corresponds to an algorithm, called *QUECA*, for "*QUE*ry for *C*onsistent *A*nswers", which, given a first order query[1] Q, generates again a new query $QUECA(Q)$, whose answers in r are consistent with IC, but as opposed to T_ω, it guarantees termination, soundness and completeness for a larger set of integrity constraints.

The implementation is done in XSB [6], a powerful logic programming system, which is provided with useful functionalities for the right implementation and operation of the consistent query answering algorithm.

In Section 2, we will show the most relevant characteristics of the operator T_ω and what makes it difficult to implement. We will also give a description of what we will understand by a database repair, query, integrity constraint and consistent answer. Next, in Section 3 we present the algorithms which generate a query $QUECA(Q)$ for a given first order query Q. In Section 4 the properties of these algorithms are analyzed, namely: scope, runtime complexity, termination, soundness and completeness. In Section 5 we describe issues regarding the implementation done in XSB. Finally, in Section 6 we draw some conclusions and

[1] Aggregate queries are being treated in [2].

propose some extensions to the solution presented in this article. Due to space restrictions we do not give proofs of propositions; we leave them for an extended version.

2 Preliminaries

2.1 Basic Notions

We start from a fixed set, *IC*, of integrity constraints associated to fixed relational database schema. We assume that *IC* is consistent. A database instance r is consistent if it satisfies *IC*, that is $r \models IC$. Otherwise, we say that r is inconsistent. If r is inconsistent, its repairs are database instances (wrt the same schema and domain) that, each of them, satisfy *IC* and differ from r by a minimal set of inserted or deleted tuples. A tuple $\bar{t}$ is a *consistent* answer to a query $Q(\bar{x})$ wrt *IC*, and we denote this with $r \models_c Q(\bar{t})$, if for every repair r' of r, $r' \models Q(\bar{t})$.

Example 2. Consider a distributors database. $\mathit{Provider}(u, v)$ means that product v is provided by u, and $\mathit{Receives}(u, v)$ that product v is received from provider u. The following ICs state that the products supplied by a provider are received from him (and vice versa), and that a provider supplies only one product.

$$\forall u, v.\ (\mathit{Provider}(u, v) \Rightarrow \mathit{Receives}(u, v))\ ,$$

$$\forall u, v.\ (\mathit{Receives}(u, v) \Rightarrow \mathit{Provider}(u, v))\ ,$$

$$\forall u, v, z.\ (\mathit{Provider}(u, v) \wedge \mathit{Provider}(u, z) \Rightarrow v = z)\ .$$

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

Provider		Receives	
a	b	a	b
a	c	a	c
d	e	d	e

has two repairs:

r' :

Provider		Receives	
a	c	a	c
d	e	d	e

r'' :

Provider		Receives	
a	b	a	b
d	e	d	e

Here, the only consistent answer to the query $\mathit{Provider}(u, v)$? in the database instance r is (d, e): $r \models_c \mathit{Provider}(d, e)$.

2.2 The T_ω Operator

The T_ω operator [1] is defined based on a previous residue calculation stage[2] which generates the necessary rules to feed the operator. Generally speaking, it is defined as a collection of operators $\mathrm{T}_0 \ldots \mathrm{T}_n$ (for some n called *finiteness point*), that were calculated based on the residues generated for that query according to the existing set *IC* of integrity constraints. The (semantical) *finiteness point* was defined as the step in which further computation (i.e. calculate T_{n+1}) had no practical sense because $\mathrm{T}_n \Rightarrow \mathrm{T}_{n+1}$. We illustrate the application of this operator by means of an example.

[2] See Sections 3 and 3.1 for a description of what residues are and how to obtain them.

Example 3. With the set of integrity constraints of example 2 and with the query $P(u,v)$, we will compute $T_\omega(P(u,v))$, letting P stand for *Provider* and R for *Receives*.

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

It seems as if T_3 is very different from T_2, however, if we rewrite them by hand we have

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

We can easily see that $\mathrm{T}_2(P(u,v)) \equiv \mathrm{T}_3(P(u,v))$, therefore the finiteness point is 2 and the modified query is $\mathrm{T}_0(P(u,v)) \wedge \mathrm{T}_1(P(u,v)) \wedge \mathrm{T}_2(P(u,v))$. □

Although operator T_ω is sound, it lacks a more general completeness result; and when thinking of a possible implementation, the termination issue is critical because the finiteness point can be very complicated to detect, even in simple examples like the one above (or may be an undecidable problem). An initial approach consisted in using Otter [5], but it turned out to be cumbersome and sometimes it did not deliver the expected results. For instance, it was not able to solve the previous example. Furthermore, even if it does work, the *offline* nature of such process makes it unsuitable for a real world implementation where a user should interact directly with the query answering system.

Thus we need to modify the previous approach to improve the results regarding termination, and possibly extending completeness as well.

In consequence, we face the problem of modifying T_ω, providing a new, more practical mechanism, but preserving the nice properties T_ω had in terms of soundness and completeness. We need to add a stronger termination property which makes the new mechanism more likely for implementation. The basic approach involves identifying a stronger syntactical condition to achieve semantically correct results.

2.3 Integrity Constraints

In this paper we will only consider only static first order integrity constraints. As in [1], we will only consider *universal* constraints that can be transformed into a standard format

Definition 1. *An integrity constraint is in standard format if it has the form*

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

where $\forall$ represents the universal closure of the formula, $\bar{x}_i$, $\bar{y}_i$ are tuples of variables, the P_i's and Q_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.

Notice that in these ICs, constants, if needed, can be pushed into ψ. Also notice that equality is allowed in ψ.

Because of implementation issues we shall negate the ICs in standard format, representing ICs as denials, that is as range restricted goals of the form

$$\leftarrow l_1 \wedge \cdots \wedge l_n \ , \tag{1}$$

where each l_i is a literal and variables are assumed to be universally quantified over the whole formula. We must emphasize the fact that this is just notation, and from now on we shall talk about of ICs assuming they are in denial form in the sense of classical logic and not of logic programming.

We shall note, however, that not all integrity constraints may be transformed into standard format, and therefore are not considered in this article. Such is the case of unsafe ICs [7], as $\forall \bar{x} \exists y.\ (P(\bar{x})) \rightarrow Q(\bar{x}, y)$.

3 Query Generation for Consistent Answers

The whole process of query rewriting for consistent answers relies on the concept of *residues* developed in the context of semantic query optimization [3]. Residues, simply put, show the interaction between an integrity constraint and a literal name[3]. Thus, a literal name which does not appear in any constraint does not have any (non–maximal [3]) residues, i.e. there are no restrictions applied to that literal. Similarly, a literal that appears more than once in an IC or set of ICs, may have several residues, which may or may not be redundant (see Definition 2).

To calculate the residues in a database schema, we will introduce Algorithm 1, which shows how to systematically obtain residues for a given literal name. Because only literal names appearing in an integrity constraint generate (non-maximal) residues, the algorithm will only be applied to them, and not to every relation in r.

Once we have calculated all the residues associated to a literal name appearing in *IC*, we shall present a second algorithm *QUECA*, that will generate the

[3] Literal names denote relations, so different literals may have the same literal name, e.g. $P(u)$ and $P(v)$ have the same literal name P. Literal names may be negative, e.g. $\neg P$, where P is a predicate name; and have an associated arity that further differentiates them (like Prolog convention), so from now on when talking about a literal, say $P(u, v)$, we are really talking about its literal name, $P/2$.

queries for consistent answers on the basis of the residues that have been already computed. We will also show how this algorithm differs from the operator T_ω presented in [1], not only in terms of termination, but in the operation itself and the necessary conditions for sound execution.

3.1 Residue Calculation

The first step in the residue calculation determines for whom they are to be calculated. In our case, it is for every literal name appearing in an integrity constraint. Because of this we must first build a list of ICs and a list of the distinct literal names L_P appearing in *IC*. This list of integrity constraints L_{IC} will only include the bodies of the ICs(represented in the form (1)). That is, given the set of integrity constraints *IC*, we build $L_{IC} = \{[l_1 \wedge \ldots \wedge l_n] \mid \forall(\leftarrow l_1 \wedge \ldots \wedge l_n) \in IC\}$. It should be noted that when negating a member of L_{IC} we obtain a clause.

Example 4. Let *IC* be the following set of integrity constraints taken from Example 2 expressed in the form (1).

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

From this we would generate $L_{IC} = \{[P(u,v) \wedge \neg R(u,v)], [\neg P(u,v) \wedge R(u,v)], [P(u,v) \wedge P(u,z) \wedge y \neq z]\}$ and $L_P = \{P(u,v), R(u,v), \neg P(u,v), \neg R(u,v)\}$. We should recall that in L_P we have the following literal names: $P/2$, $R/2$, $\neg P/2$ and $\neg R/2$. □

Next, to calculate the residues coming from $l \in L_P$, and $ic \in L_{IC}$, we use the subsumption algorithm presented in [3]. However, because we are dealing with an implementation, we need a systematical procedure to obtain residues. The method utilized is formalized as Algorithm 1.

Example 5. (Example 4 Continued) Applying Algorithm 1 up to line 13, to $l = P(u,v)$ and every member of L_{IC}, we would obtain one residue for each occurrence of $P/2$: $residue_1(P(u,v)) := R(u,v)$, $residue_2(P(u,v)) := \neg P(u,z) \vee v = z$ and $residue_3(P(u,v)) := \neg P(u,w) \vee w = v$. Here we may find redundant residues (see Definition 2). □

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, we take care of eliminating redundant residues as we build the conjunction (steps 14–21 in Algorithm 1) in order to reduce complexity in the following phase ($QUECA$).

Definition 2 (Residue Redundancy). *Let $R \wedge \varphi$ be a conjunction of residues associated to a literal l, where R is a clause and φ a conjunction of clauses. We will say R is redundant in $R \wedge \varphi$ if exists a clause $R' \in \varphi$ and a substitution $\sigma : (Var(R')$[4]$\setminus Var(l)) \rightarrow (Var(R) \setminus Var(l))$, such that $R'\sigma \equiv R$.*

[4] Var(X) is the set of all (quantified or unquantified) variables in the expression X.

Algorithm 1 Compute *residues*(l)

```
Require: Set of integrity constraints in denial form $IC$.
Ensure: $residues(l)$ is a formula in CNF that contains all the residues associated to a
    literal $l$.
 1: Create list $L_{IC}$ of integrity constraint bodies and a list $L_P$ of distinct literal names
    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
```

Note that, in the definition above, if R is redundant in $R \wedge \varphi$, then $R \wedge \varphi$ is logically equivalent to φ. The elimination of redundant residues is based on unification and is done in steps 14–21 of Algorithm 1.

Example 6. (Example 5 Continued) By Definition 2, we have that $residue_3(P(u,v))$ is a redundant residue, because there exists a substitution $\sigma : \ z \mapsto w$, such that $residue_2(P(u,v))\sigma = residue_3(P(u,v))$. Thus, we have $residues(P(u,v)) = [R(u,v)] \wedge [\neg P(u,z) \vee v = z]$. □

Note that the definition does not state that it detects *all* redundancies, but only those subject to the sufficient condition presented. For example, if we consider the following residues for $R(x)$: $residue_1(R(x)) = P(x) \vee x > 100$ and $residue_2(R(x)) = P(x) \vee x > 50$. Clearly $residue_1$ includes the information in $residue_2$, so $residue_2$ would be redundant; However, Definition 2 does not detect it. This occurs mainly when ICs are redundant, which can easily be avoided for cases like these. As shown in Example 6, functional dependencies are a common case of ICs which generate redundant residues according to Definition 2. The reason why residue redundancy is not treated further is due to the complexity of implementation, which could be far higher than the performance improvement we could get in the next stage (*QUECA*). Besides, residue redundancy can become such a large subject that it would deviate the central point of attention of this article, which is to build the queries for consistent answers.

Example 7. Finally, by applying Algorithm 1 to the set *IC* presented in Example 4, we would obtain:

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

3.2 Query Generation (*QUECA*)

Once all the residues have been computed, and given a query Q, we can generate the query, $QUECA(Q)$, which will deliver consistent answers from a consistent or inconsistent database. This query differs from Q only when Q has residues, so $QUECA(Q)$ should be only executed for literal names appearing in *IC*.

Initially the query $QUECA(Q)$ is equal to Q, plus a list of pending residues which are the residues associated to Q calculated by Algorithm 1.[5] 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. This condition is, informally, 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. We will see later that the procedure does not always terminate.

Example 8. Consider the following hypothetical pairs of queries and residues:

$$\begin{array}{llll} Query: & 1.\ S(u) & Residues: & S(u) \\ & 2.\ M(u) & & N(u) \\ & 3.\ P(u,v) & & \forall z\ (P(u,v) \vee \neg Q(u,z)) \ . \end{array}$$

Clearly in the first case, the residue can be discarded because 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

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

□

This method works when only conjunctions are involved (case 2 in Example 8), because determining if a residue should be part of the query or not is easy. However, most of the residues are clauses (case 3 in Example 8), so we must somehow deal with disjunction.

The way to solve this problem is by keeping conjunctions together, i.e. working in DNF. To do so, when a clausal residue adds new information to a query, we make as many copies of the query as literals in the residue we are adding,

[5] The residues are in CNF, we will treat every clause as an element of a list.

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 plus the residues coming from the newly appended literal. We shall informally call this a split operation. These copies, connected together by disjunctions, would constitute the final query $QUECA(Q)$.

Example 9. (Example 8 Continued) In the third case of the previous example we would then have

$$\begin{array}{ll}
\textit{Query}: & \textit{Residues}: \\
\quad 3.\ E_1 : P(u,v) \wedge P(u,v) & \quad R_1 : \textit{residues}(P(u,v)) \\
\quad\quad E_2 : P(u,v) \wedge \neg Q(u,z) & \quad R_2 : \textit{residues}(\neg Q(u,z))\ .
\end{array}$$

So, we have $QUECA(Q) = \forall z(E_1 \vee E_2)$ where E_i is a disjunctionless formula, and $\textit{Residues} = R_1, R_2$, where each R_i belongs to its corresponding E_i. □

This clarifies the need for a new notation that will enable us to keep track of the residues involved in building each E. Furthermore, this notation should not only include the literals in E and its associated Pending Residue List, but it should also "remember" the last residue that provoked one of these split operations, in order to avoid inserting a residue whose information was already inserted earlier. For these purposes we define a *Temporary Query Unit* (TQU).

Definition 3. *A temporary query unit, $D : E \bullet R$, consists of a set of clauses D, a conjunction of literals E and a conjunction of residues R.*

Both symbols, : and $\bullet$, are only used to separate D, E and R from each other. D represents the last residues involved in building E and R is the conjunction of residues $\phi_1 \wedge \cdots \wedge \phi_n$ yet to be resolved. We shall note that all variables coming from a residue appear universally quantified in D and E(see Example 10). Both symbols have higher precedence that any other connective. In this way, $QUECA(Q)$ can be seen as a disjunction of temporary query units, $\bigvee TQU$, when we reach the point in which $R = \varnothing$ for every TQU.

Example 10. (Example 9 Continued) Using the new notation for the third case we would have:

$QUECA(P(u,v))$:

$$\begin{array}{ll}
TQU_1 & [\forall z(P(u,v) \vee \neg Q(u,z))] : P(u,v) \wedge P(u,v) \bullet \textit{residues}(P(u,v)) \ \vee \\
TQU_2 & \underbrace{[\forall z(P(u,v) \vee \neg Q(u,z))]}_{D} : \underbrace{P(u,v) \wedge \forall z \neg Q(u,z)}_{E} \bullet \underbrace{\textit{residues}(\neg Q(u,z))}_{R}\ .
\end{array}$$

□

The critical step is then determining when a residue 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 when $E \models \phi_1$[6] or $D \models \phi_1$, then ϕ_1 can be discarded. If either condition is not satisfied, the residue must be included in the query.[7] In example 10, we have from TQU_1 that $D \models \textit{residues}(P(u))$, thus they can be

[6] The required condition is that every term in ϕ_1 belongs to E.
[7] We will see that sometimes only part of the residue must be included.

discarded and the iteration would have ended for TQU_1. This is the semantic result we want to obtain via syntactical means. The usual way to attain this is via unification.

In our case we will define a sort of one way unification in which only certain types of variables will be involved: New Variables in a TQU and Free Variables in a Residue.

Definition 4. *A New Variable in a $TQU = D : E \bullet R$ associated to a query Q is a variable that belongs to $newVar(TQU) := Var(E) \setminus Var(Q)$ and is universally quantified.*

Definition 5. *Given a $TQU = D : E \bullet R$, a Free Variable in a Residue $\phi \in R$ is a variable 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. For instance, in example 10, 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 than in that residue.

We can now formally define the meaning of *the information of a residue already in a TQU.*

Definition 6. *We will say the information of a residue $\phi = l_1 \vee \cdots \vee l_n$ is already in a $TQU = D : E \bullet R$, and will write $\phi \mathrel{\widetilde{\in}} D : E$, whenever there exists a substitution $\sigma : freeVar(\phi) \to newVar(TQU) \cup freeVar(D)$, such that $\phi\sigma \in D$ or for all i, $l_i\sigma \in E$. In case only some $l_i\sigma \in E$, we will say the information of a residues is already partially in a TQU, and we will write $\phi \;{}_p\widetilde{\in}_\sigma\; D : E$.*

Notice that if $freeVar(\phi) = \varnothing$, then σ could be ε (the identity).

Consequently we have that, when verifying whether to add a residue $\phi = l_1 \vee \cdots \vee l_m$ to a query, if $\phi \mathrel{\widetilde{\in}} D : E$, then ϕ is discarded. Otherwise, it must be added to E and one of the mentioned split operations must take place. However, if $\phi \mathrel{\widetilde{\in}_{P,\theta}} D : E$, then we must keep a copy of $D : E \bullet R$; 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 and its residues must be added at the end of a copy R_i of R.

Example 11. (Example 10 Continued) By using the method presented above the second $P(u, v)$ would not be included in TQU_1, that is

$QUECA(P(u, v))$:

$$\begin{array}{ll} TQU_1 & [P(u,v) \vee \neg Q(u,z)] : P(u,v) \bullet residues(P(u,v)) \;\vee \\ TQU_2 & [P(u,v) \vee \neg Q(u,z)] : P(u,v) \wedge \forall z \neg Q(u,z) \bullet residues(\neg Q(u,z)) \;. \quad \square \end{array}$$

The procedure just described is formalized in Algorithm 2.

Example 12. (Example 3 Continued) We will show how Algorithm 2 computes $QUECA(P(u, v))$, which is equivalent to $\mathrm{T}_2(P(u, v))$, being 2 the finiteness point.

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

By rearranging the result by hand, we obtain

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

and we can see how the constraints get spread towards the related literals, in this case $R/2$, where we can see how the functional dependency of the second argument of $P/2$ has generated a functional dependency for the second argument of $R/2$ due to the nature of *IC*. This example was shown to be non terminating for T_ω (see Example 3), but is now solved by *QUECA*. □

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

```
Require: Algorithm 1 has been executed.
Ensure: $QUECA(l)$ contains the expected results.
 1: $QUECA(l) := \varnothing$
 2: $TQUs := \varnothing : l \bullet residues(l)$
 3: while $TQUs \neq \varnothing$ do
 4:   select(extract) first $TQU$ from $TQUs \mapsto (D\colon E \bullet R)$
 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:       $TQUs = D\colon E \bullet R \vee TQUs$
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:       $TQUs := \bigvee_{i=0}^{m}(D_i\colon E_i \bullet R_i) \vee TQUs$
29:     end if
30:   end if
31: end while
```

In the previous example we can 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 occurs when the $\bullet$ reaches the end of R, put in other words, when no residues are left to be resolved.

4 Properties of *QUECA*

In this section we will show that *QUECA* algorithm is well behaved for an interesting syntactical class of ICs.

Definition 7. *(a) A* binary integrity constraint *(BIC) is a denial of the form* $\forall\ (\leftarrow l_1(\bar{x}_1) \wedge l_2(\bar{x}_2) \wedge \psi(\bar{x}))$*, where* l_1 *and* l_2 *are database literals, and* ψ *is a formula that only contains built-in predicates.*
(b) A set of BICs, IC, is fact-oriented[8] *if there is a tuple* $\bar{a}$ *and a literal name* L*, such that* $IC \models L(\bar{a})$*.*

Usually ICs are not fact-oriented. As a particular case of BICs, we obtain *unary integrity constraints*, which have just one database literal and possibly a formula with built-in predicates. In the class of binary contraints we find functional dependencies, inclusion dependencies, symmetry constraints, and domain and range constraints. In consequence, we are covering most of the static constraints found in traditional relational databases, excluding (existential) referential ICs, transitivity constraints, and possibly other constraints that might be better expressed as rules or views at the application layer.

The following results apply to the case of a finite set of BICs.

Theorem 1. *The worst case runtime complexity of Algorithm 1 for residue computation is* $O(n^2)$*, where* n *represents the number of ICs.*

Theorem 2 (Termination). *Given a set of non fact-oriented binary integrity constraints, Algorithm 2 terminates in a finite number of steps.*

The termination property is based on the fact that by restricting execution to BICs only, residues contain one literal name at most, which in the worst case generates an infinite sequence of single literals. The infiniteness of this sequence is then limited by the condition in line 12 of Algorithm 2 and the fact that we only consider range–restricted ICs (1), conditions which ensure that at a given point, pending residues add no new information to the resulting query, thus being discarded. Notice that this result extends the termination results presented in [1], where semantic termination was only ensured for uniform binary constraints.

Theorem 3. *For non fact-oriented binary ICs, and a literal name* L*, the worst case runtime complexity of Algorithm 2 running on* L *is* $O(nk^{8n})$*, where* n *represents the number of ICs and* k *is the maximum number of terms per integrity constraint.*

Although this is not an encouraging result, we will see in Section 5 that this process is done at compile–time, so it should not affect the performance from a user's point of view.

[8] Fact-oriented integrity constraints can be seen as a special case of tuple-generating dependencies, *tgd's* [7], in which the body may contain equality. A common fact-oriented constraint is of the form *true* $\Rightarrow L(a)$.

It is possible to prove that the *QUECA* algorithm can simulate the iterative application of operator T until the point where *QUECA* stops. At that point we obtain a corresponding semantical termination point for T. The main difference is that, while T would perform 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, *QUECA* does this on a per-residue basis. This eliminates residues one by one, thus obtaining a much more efficient query (see the difference between $T_3(P(u,v))$ in Example 3 and $QUECA(P(u,v))$ in Example 12). Having mapped *QUECA*'s execution to that of T, we may take advantage of soundeness and completeness results for T.

Theorem 4 (Soundness). *Let r be a database instance, IC a set of binary integrity constraints and $Q(\bar{x})$ a literal query, such that $r \vDash QUECA(Q(\bar{t}))$. If Q is universal or non-universal and domain independent, then $\bar{t}$ is a consistent answer to Q in r, that is, $r \vDash_c Q(\bar{t})$.*

Theorem 5 (Completeness). *Let r be a database instance and IC a set of non fact-oriented binary integrity constraints, then for every ground literal $l(\bar{t})$, if $r \vDash_c l(\bar{t})$, then $r \vDash QUECA(\bar{t})$.*

All the results above can be easily extended to queries that are conjunctions of literals without existential quantifiers.

5 Implementation

To achieve the objectives of this work we need a common framework for data, rules, queries and integrity constraints, to be able to perform operations on them and elaborate the queries for consistent answers mentioned earlier. Logic Programming languages provide this framework and XSB seems an adequate candidate. Generally speaking we prefer an LP language because the algorithms presented in this article need the ability to perform unifications, substitutions and detecting subsumption. Perhaps what makes XSB a better candidate that other LP languages is, apart from the Relational DMBS interface, Foreign Language interface and the fact it runs on multiple platforms, its *tabling* capabilities that improve its efficiency over other systems that would, for example, have to recalculate the residues every time they are needed by Algorithm 2.

Our system consists of a four modules which provide several predicates that allow the user direct interaction with the system. Upon initialization, the program connects itself to a database previously defined by the user, executes both algorithms presented in this paper and stores their results on XSB's tables. This avoids having to recalculate residues, *QUECAs* or their equivalent SQL strings, thus practically eliminating the relevance of the exponential runtime complexity of Algorithm 2.

The integrity constraints of the form (1) are read from the file named `ics`, in which they are written with the following syntax:

```
<- [ ... denials ... ].
```

For instance, to include the ICs corresponding to Example 12, we would modify the file `ics` to contain:

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

Once initialization is over, the user may query directly the database or retrieve one of the computed residues, *QUECAs* or SQL strings. For example, by executing `| ?-queca(p(X,Y),Q).` we would obtain:

```
Q = and(p(id1,id2),all(u1,and(r(id1,id2),or(and(no(p(id1,u1)),
    no(r(id1,u1))),equal(id2,u1)))))
```

With this method we can answer any query that is free of disjunctions and existential quantifiers. Due to space limitations, further details of the implementation are included in an extended version of this paper.

6 Conclusions

We have shown an algorithm to obtain consistent answers to queries posed to inconsistent databases. This algorithm is proved to be terminating, sound and complete for the class of non fact-oriented binary ICs. The termination results extends those obtained in [1].

We also implemented the algorithm on XSB with a program that interfaces directly to a given RDBMS. The next steps towards an effective application include handling queries and integrity constraints with existential quantifiers. Complete elimination of residue redundancy could be addressed as well.

Acknowledgements

This work has been supported by Fondecyt Grants (# 1980945 and # 1000593). We are grateful to Marcelo Arenas for illuminating conversations.

References

1. M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *Proceedings ACM Symposium on Principles of Database Systems (ACM PODS '99, Philadelphia)*, pages 68–79. ACM Press, 1999.
2. M. Arenas, L. Bertossi, and J. Chomicki. Aggregation in Inconsistent Databases. In Preparation, 2000.
3. U.S Chakravarthy, John Grant, and Jack Minker. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.
4. S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26:65–74, March 1997.
5. W.W. McCune. *OTTER 3.0 Reference Manual and Guide.* Argonne National Laboratory, Technical Report ANL-94/6, 1994.
6. K. F. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of SIGMOD 1994 Conference*, pages 442–453. ACM Press, 1994.
7. J.D. Ullman. *Principles of Database and Knoledge-Base Systems*, volume I. Computer Science Press, Maryland, 1988.