In: Proc. 19th International Conference on Logic Programming (ICLP 2003), LNCS,

Efficient Evaluation of Logic Programs for Querying Data Integration Systems*

Thomas Eiter[1], Michael Fink[1], Gianluigi Greco[2], and Domenico Lembo[3]

[1] Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria
eiter@kr.tuwien.ac.at, michael@kr.tuwien.ac.at
[2] DEIS University of Calabria, Via Pietro Bucci 41C, I-87036 Rende, Italy
ggreco@si.deis.unical.it
[3] DIS University of Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy
lembo@dis.uniroma1.it

Abstract. Many data integration systems provide transparent access to heterogeneous data sources through a unified view of all data in terms of a global schema, which may be equipped with integrity constraints on the data. Since these constraints might be violated by the data retrieved from the sources, methods for handling such a situation are needed. To this end, recent approaches model query answering in data integration systems in terms of nonmonotonic logic programs. However, while the theoretical aspects have been deeply analyzed, there are no real implementations of this approach yet. A problem is that the reasoning tasks modeling query answering are computationally expensive in general, and that a direct evaluation on deductive database systems is infeasible for large data sets. In this paper, we investigate techniques which make user query answering by logic programs effective. We develop pruning and localization methods for the data which need to be processed in a deductive system, and a technique for the recombination of the results on a relational database engine. Experiments indicate the viability of our methods and encourage further research of this approach.

1 Introduction

Data integration is an important problem, given that more and more data are dispersed over many data sources. In a user friendly information system, a data integration system provides transparent access to the data, and relieves the user from the burden of having to identify the relevant data sources for a query, accessing each of them separately, and combining the individual results into the global view of the data.

Informally, a data integration system $\mathcal{I}$ may be viewed as system $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ that consists of a global schema $\mathcal{G}$, which specifies the global (user) elements, a source schema $\mathcal{S}$, which describes the structure of the data sources in the system, and a mapping $\mathcal{M}$, which specifies the relationship between the sources and the global schema. There are basically two approaches for specifying the mapping [14]: the global-as-view (GAV) approach, in which global elements are defined as views over the sources, and the local-as-view (LAV), in which conversely source elements are characterized

* This work has been partially be supported by the European Commission FET Programme Projects IST-2002-33570 INFOMIX and IST-2001-37004 WASP.

$player^{\mathcal{R}}$:

10	*Totti*	*RM*
9	*Beckham*	*MU*

$team^{\mathcal{R}}$:

RM	*Roma*	10
MU	*Man. Utd.*	8
RM	*Real Madrid*	10

$coach^{\mathcal{R}}$:

7	*Ferguson*	*MU*

Fig. 1. Global database for the football scenario as retrieved from the sources.

as views over the global schema. In this paper, we concentrate on the GAV approach which is in general considered appropriate for practical purposes.

Usually, the global schema also contains information about constraints, Σ, such as key constraints or exclusion dependencies issued on a relational global schema. The mapping $\mathcal{M}$ is often defined by views on the sources $\mathcal{S}$, in a language which amounts to a fragment of stratified Datalog.

Example 1. As a running example, we consider a data integration system $\mathcal{I}_0 = \langle \mathcal{G}_0, \mathcal{S}_0, \mathcal{M}_0 \rangle$, referring to the context of football teams. The global schema $\mathcal{G}_0$ consists of the relation predicates $player(Pcode, Pname, Pteam)$, $team(Tcode, Tname, Tleader)$, and $coach(Ccode, Cname, Cteam)$. The associated constraints Σ_0 are that the keys of $player$, $team$, and $coach$, are the attributes $Pcode$, $Tcode$, and $Ccode$ respectively, and that a coach can neither be a player nor a team leader. The source schema $\mathcal{S}_0$ comprises the relations s_1, s_2, s_3 and s_4. Finally, the mapping $\mathcal{M}_0$ is defined by the datalog program $player(X,Y,Z) \leftarrow s_1(X,Y,Z,W)$; $team(X,Y,Z) \leftarrow s_2(X,Y,Z)$; $team(X,Y,Z) \leftarrow s_3(X,Y,Z)$; $coach(X,Y,Z) \leftarrow s_4(X,Y,Z)$. $\Box$

When the user issues a query q on the global schema, the global database is constructed by data retrieval from the sources and q is answered from it. However, the global database might be inconsistent with the constraints Σ.

Example 2. Suppose the query $q(X) \leftarrow player(X,Y,Z)$; $q(X) \leftarrow team(V,W,X)$ is issued in our scenario, which asks for the codes of all players and team leaders. Assuming that the information content of the sources is given by the database $\mathcal{D}_0 = \{\ s_1(10, \mathit{Totti}, \mathit{RM}, 27)$, $s_1(9, \mathit{Beckham}, \mathit{MU}, 28)$, $s_2(RM, \mathit{Roma}, 10)$, $s_3(MU, \mathit{Man.\ Utd.}, 8)\}$, $s_3(RM, \mathit{Real\ Madrid}, 10)\}$ $s_4(7, \mathit{Ferguson}, MU)\}$, the global database $\mathcal{R}$ in Fig. 1 is constructed from the retrieved data. It violates the key constraint on $team$, witnessed by the facts $team(RM, \mathit{Roma}, 10)$ and $team(RM, \mathit{Real\ Madrid}, 10)$, which coincide on $Tcode$ but differ on $Tname$. $\Box$

To remedy this problem, the inconsistency might be eliminated by modifying the database and reasoning on the "repaired" database. The suitability of a possible repair depends on the underlying semantic assertions which are adopted for the database; in general, not a single but multiple repairs might be possible [2, 6].

Recently, several approaches to formalize repair semantics by using logic programs have been proposed [3, 12, 4, 7, 5]. The common basic idea is to encode the constraints Σ of $\mathcal{G}$ into a logic program, Π, using unstratified negation or disjunction, such that the stable models of this program yield the repairs of the global database. Answering a user query, q, then amounts to cautious reasoning over the logic program Π augmented with the query, cast into rules, and the retrieved facts $\mathcal{R}$.

An attractive feature of this approach is that logic programs serve as executable logical specifications of repair, and thus allow to state repair policies in a declarative manner rather than in a procedural way. However, a drawback of this approach is that with current implementations of stable model engines, such as DLV or Smodels, the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability. This calls for suitable optimization methods that help in speeding up the evaluation of queries expressed as logic programs [5].

In this paper, we face this problem and make the following contributions:

(1) We present a basic formal model of data integration via logic programming specification, which abstracts from several proposals in the literature [3, 12, 4, 7, 5]. Results which are obtained on this model may then be inherited by the respective approaches.
(2) We foster a *localization approach* to reduce complexity, in which irrelevant rules are discarded and the retrieved data is decomposed into two parts: facts which will possibly be touched by a repair and facts which for sure will be not. The idea which is at the heart of the approach is to reduce the usage of the nonmonotonic logic program to the essential part for conflict resolution. This requires some technical conditions to be fulfilled in order to make the part "affected" by a repair small (ideally, as much as possible).
(3) We develop *techniques for recombining* the decomposed parts for query answering, which interleave logic programming and relational database engines. This is driven by the fact that database engines are geared towards efficient processing of large data sets, and thus will help to achieve scalability. To this end, we present a marking and query rewriting technique for compiling the reasoning tasks which emerge for user query evaluation into a relational database engine.

In our overall approach, the attractive features of a nonmonotonic logic programming system, such as DLV [15], can be fruitfully combined with the strengths of an efficient relational database engine. The experimental results are encouraging and show that this combination has potential for building advanced data integration systems with reasonable performance. Moreover, our main results might also be transferred to other logic programming systems (e.g., Smodels [17], possibly interfaced by XSB [20], or Dislog [19]).

For space reason, the exposition is necessarily succinct and some details are omitted. They are provided in an extended version of the paper [9].

2 Preliminaries

A *Datalog*$^{\vee,\neg}$ *rule* r is a clause $a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_{k+m}$ where $n > 0$, $k, m \geq 0$ and $a_1, \ldots, a_n, b_1, \ldots, b_{k+m}$ are function-free atoms. If $n = 1$ then r is a *Datalog*$^{\neg}$ *rule*, and if also $k + m = 0$ the rule is simply called *fact*. If $m = 0$ and $n = 1$, then r is a *plain Datalog* (or simply, a Datalog) *rule*. A *Datalog*$^{\vee,\neg}$ *program* $\mathcal{P}$ is a finite set of Datalog$^{\vee,\neg}$ rules; it is *plain* (or simply, a Datalog program), if all its rules are plain. A Datalog$^{\neg}$ program with stratified negation is denoted by *Datalog*$^{\neg_s}$.

The model theoretic semantics assigns to any program $\mathcal{P}$ the set of its (disjunctive) *stable models* [11, 18], denoted by $\mathrm{SM}(\mathcal{P})$. As well-known, $|\mathrm{SM}(\mathcal{P})| = 1$ for any plain and Datalog$^{\neg_s}$ program $\mathcal{P}$. Given a set of facts $\mathcal{D}$, the program $\mathcal{P}$ on input $\mathcal{D}$, denoted by $\mathcal{P}[\mathcal{D}]$, is the union $\mathcal{P} \cup \mathcal{D}$. For further background, see [11, 18, 10].

Databases. We assume a finite, fixed database domain $\mathcal{U}$ whose elements are referenced by constants c_1,..., c_n under the *unique name assumption*, i.e., different constants denote different objects (we will briefly address infinite domains at the end of Section 4). A *relational schema* (or simply *schema*) $\mathcal{RS}$ is a pair $\langle \Psi, \Sigma \rangle$, where:

- Ψ is a set of relation (predicate) symbols, each with an associated arity that indicates the number of its attributes;
- Σ is a set of *integrity constraints* (ICs), i.e., a set of assertions that are intended to be satisfied by each database instance. We assume to deal with universally quantified constraints [1], i.e., first order formulas of the form:

$$\forall(\boldsymbol{x}) \bigwedge_{i=1}^{l} A_i \supset \bigvee_{j=1}^{m} B_j \vee \bigvee_{k=1}^{n} \phi_k, \qquad (1)$$

  where $l+m > 0, n \geq 0$, $\boldsymbol{x}$ is a list of distinct variables, $A_1, \ldots, A_l$ and $B_1, \ldots, B_m$ are positive literals, and $\phi_1, \ldots, \phi_n$ are built-in literals.

Notice that many classical constraints issued on a relational schema can be expressed in this format, as key and functional dependencies, exclusion dependencies, or inclusion dependencies of the form $\forall(\boldsymbol{x}) p_1(\boldsymbol{x}) \supset p_2(\boldsymbol{x})$.

Example 3. In our ongoing example, the global schema $\mathcal{G}_0$ is the database schema $\langle \Psi_0, \Sigma_0 \rangle$, where Ψ_0 consists of the ternary relation symbols $player$, $team$, and $coach$, and Σ_0 can be formally defined as follows (quantifiers are omitted):

$$\begin{array}{rr}
player(X,Y,Z) \wedge player(X,Y1,Z1) \supset Y{=}Y1; & player(X,Y,Z) \wedge player(X,Y1,Z1) \supset Z{=}Z1 \\
team(X,Y,Z) \wedge team(X,Y1,Z1) \supset Y{=}Y1; & team(X,Y,Z) \wedge team(X,Y1,Z1) \supset Z{=}Z1 \\
coach(X,Y,Z) \wedge coach(X,Y1,Z1) \supset Y{=}Y1; & coach(X,Y,Z) \wedge coach(X,Y1,Z1) \supset Z{=}Z1 \\
coach(X,Y,Z) \wedge player(X1,Y1,Z1) \supset X{\neq}X1; & coach(X,Y,Z) \wedge team(X1,Y1,Z1) \supset X{\neq}Z1
\end{array}$$

The first three rows encode the key dependencies, whereas the last row models the two constraints stating that a coach cannot be a player or a team leader. $\square$

A *database instance* (or simply *database*) $\mathcal{DB}$ for a schema $\mathcal{RS}$ is a set of facts of the form $r(t)$ where r is a relation of arity n in Ψ and t is an n-tuple of values from $\mathcal{U}$.

Given a constraint $\sigma \in \Sigma$ of the form $\forall(\boldsymbol{x})\alpha(\boldsymbol{x})$, we denote by $ground(\sigma)$ the set of all ground formulas $\alpha(\boldsymbol{x})\theta$, also called *ground constraints*, where θ is any substitution of values in $\mathcal{U}$ for $\boldsymbol{x}$; furthermore, $ground(\Sigma) = \bigcup_{\sigma \in \Sigma} ground(\sigma)$.

Given a schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ and a database instance $\mathcal{DB}$ for $\mathcal{RS}$, we say that $\sigma^g \in ground(\Sigma)$ is *satisfied* (resp., *violated*) in $\mathcal{DB}$, if σ^g evaluates to true (resp., false) in $\mathcal{DB}$. Moreover, $\mathcal{DB}$ is *consistent* with Σ if every $\sigma \in ground(\Sigma)$ is satisfied in $\mathcal{DB}$.

3 A Logic Framework for Query Answering

In this section, we present an abstract framework for modeling query answering in data integration systems using logic programs. We first adopt a more formal description of data integration systems, and then we discuss how to compute consistent answers for a user query to a data integration system where the global database, constructed by retrieving data from the sources, might be inconsistent.

3.1 Data Integration Systems

Formally, a data integration system $\mathcal{I}$ is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where:

1. $\mathcal{G}$ is the *global schema*. We assume that $\mathcal{G}$ is a relational schema, i.e., $\mathcal{G} = \langle \Psi, \Sigma \rangle$.
2. $\mathcal{S}$ is the *source schema*, constituted by the schemas of the various sources that are part of the data integration system. We assume that $\mathcal{S}$ is a relational schema of the form $\mathcal{S} = \langle \Psi', \emptyset \rangle$, i.e., there are no integrity constraints on the sources.
3. $\mathcal{M}$ is the *mapping between* $\mathcal{G}$ *and* $\mathcal{S}$. In our framework the mapping is given by the GAV approach, i.e., each global relation in Ψ is associated with a *view*, i.e., a query, over the sources. The language used to express queries in the mapping is Datalog$^{\neg_s}$.

We call any database $\mathcal{D}$ for the source schema $\mathcal{S}$ a *source database* for $\mathcal{I}$. Based on $\mathcal{D}$, it is possible to compute database instances for $\mathcal{G}$, called *global databases* for $\mathcal{I}$, according to the mapping specification. Given a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ and a source database $\mathcal{D}$, the *retrieved global database*, $ret(\mathcal{I}, \mathcal{D})$, is the global database obtained by evaluating each view of the mapping $\mathcal{M}$ over $\mathcal{D}$.

Notice that $ret(\mathcal{I}, \mathcal{D})$ might be inconsistent with respect to Σ, since data stored in local and autonomous sources need in general not satisfy constraints expressed on the global schema. Hence, in case of constraint violations, we cannot conclude that $ret(\mathcal{I}, \mathcal{D})$ is a "legal" global database for $\mathcal{I}$ [14]. Following a common approach in the literature on inconsistent databases [2, 12, 6], we define the semantics of a data integration system $\mathcal{I}$ in terms of *repairs* of the database $ret(\mathcal{I}, \mathcal{D})$.

Repairs. Let us first consider the setting of a single database. Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema, $\mathcal{DB}$ be a (possibly inconsistent) database for $\mathcal{RS}$, and $\mathcal{R}_1$ and $\mathcal{R}_2$ be two databases for $\mathcal{RS}$ consistent with Σ. We say that $\mathcal{R}_1 \leq_{\mathcal{D}} \mathcal{R}_2$ if $\triangle(\mathcal{R}_1, \mathcal{D}) \subseteq \triangle(\mathcal{R}_2, \mathcal{D})$, where $\triangle(X, Y)$ denotes the symmetric difference between sets X and Y. Furthermore, $\mathcal{R}_1 <_{\mathcal{D}} \mathcal{R}_2$ stands for $\mathcal{R}_1 \leq_{\mathcal{D}} \mathcal{R}_2 \wedge \mathcal{R}_2 \not\leq_{\mathcal{D}} \mathcal{R}_1$. Then, a database $\mathcal{R}$ is a *repair for* $\mathcal{DB}$ *w.r.t.* Σ, if $\mathcal{R}$ is a database for $\mathcal{RS}$ consistent with Σ and $\mathcal{R}$ is *minimal w.r.t.* $\leq_{\mathcal{D}}$, i.e., there exists no database $\mathcal{R}'$ for $\mathcal{RS}$ consistent with Σ such that $\mathcal{R}' <_{\mathcal{D}} \mathcal{R}$. We refer to the set of all such repairs as $rep(\mathcal{DB})$ *wrt* Σ; when clear from the context, Σ is omitted.

Definition 1. Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. A global database $\mathcal{R}$ for $\mathcal{I}$ is a *repair for* $\mathcal{I}$ *w.r.t.* $\mathcal{D}$ if $\mathcal{R}$ is a repair for $ret(\mathcal{I}, \mathcal{D})$ w.r.t. Σ. The set of all repairs for $\mathcal{I}$ w.r.t. $\mathcal{D}$ is denoted by $rep_{\mathcal{I}}(\mathcal{D})$. □

Intuitively, each repair is obtained by properly adding and deleting facts from $ret_{\mathcal{I}}(\mathcal{D})$ in order to satisfy constraints in Σ, as long as we "minimize" such additions and deletions.

Note that, in the above definition we have considered the mapping $\mathcal{M}$ as *exact*, i.e., we have assumed that the data retrieved from the sources by the views of the mapping are exactly the data that satisfy the global schema, provided suitable repairing operations. Other different assumptions can be adopted on the mapping (e.g., *soundness* or

$player^{\mathcal{R}_1}$:

10	*Totti*	*RM*
9	*Beckham*	*MU*

$team^{\mathcal{R}_1}$:

RM	*Roma*	10
MU	*Man. Utd.*	8

$coach^{\mathcal{R}_1}$:

7	*Ferguson*	MU

$player^{\mathcal{R}_2}$:

10	*Totti*	*RM*
9	*Beckham*	*MU*

$team^{\mathcal{R}_2}$:

MU	*Man. Utd.*	8
RM	*Real Madrid*	10

$coach^{\mathcal{R}_2}$:

7	*Ferguson*	MU

Fig. 2. Repairs of $\mathcal{I}_0$ w.r.t. $\mathcal{D}_0$.

completeness assumptions [14]). Roughly speaking, such assumptions impose some restrictions or preferences on the possibility of adding or removing facts from $ret(\mathcal{I}, \mathcal{D})$ to repair constraint violations, leading to different notions of minimality (see, e.g., [6, 8]).

We stress that dealing only with exact mappings is not an actual limitation for the techniques presented in the paper; in fact, in many practical cases, the computation of the repairs under other mapping assumptions can be modeled by means of a logic program similar to the computation of repairs under the exactness assumption.

Query. A *query* over the global schema $\mathcal{G}$ is a non-recursive Datalog$^{\neg}$ program that is intended to extract a set of tuples over $\mathcal{U}$; note that in real integration applications, typically a language subsumed by non-recursive Datalog is adopted. For any query $q(X_1, \ldots, X_n)$, we call the set $ans(q, \mathcal{I}, \mathcal{D}) = \{q(c_1, \ldots, c_n) \mid q(c_1, \ldots, c_n) \in \mathrm{SM}(q[\mathcal{R}])$ for each $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ $\}$ the *consistent answers* to q.

Example 4. Recall that in our scenario, the retrieved global database $ret(\mathcal{I}_0, \mathcal{D}_0)$ shown in Figure 1 violates the key constraint on $team$, witnessed by $team(RM, \mathit{Roma}, 10)$ and $team(RM,\ \mathit{Real\,Madrid}, 10)$. A repair results by removing exactly one of these facts; hence, $rep_{\mathcal{I}_0}(\mathcal{D}_0) = \{\mathcal{R}_1, \mathcal{R}_2\}$, where $\mathcal{R}_1$ and $\mathcal{R}_2$ are as shown in Figure 2. For the query $q(X) \leftarrow \mathit{player}(X, Y, Z);\ q(X) \leftarrow \mathit{team}(V, W, X)$, we thus obtain that $ans(q, \mathcal{I}_0, \mathcal{D}_0) = \{q(8), q(9), q(10)\}$. If we consider the query $q'(Y) \leftarrow \mathit{team}(X, Y, Z)$, we have that $ans(q', \mathcal{I}_0, \mathcal{D}_0) = \{q'(\mathit{Man.\ Utd.})\}$, while considering $q''(X, Z) \leftarrow \mathit{team}(X, Y, Z)$, we have that $ans(q'', \mathcal{I}_0, \mathcal{D}_0) = \{q''(\mathit{RM}, 10), q''(\mathit{MU}, 8)\}$. □

3.2 Logic Programming for Consistent Query Answering

We now describe a generic logic programming framework for computing consistent answers to queries posed to a data integration system in which inconsistency possibly arises.

According to several proposals in the literature [13, 4, 7, 5], we provide answers to user queries by encoding the mapping assertions in $\mathcal{M}$ and the constraints in Σ in a Datalog program enriched with unstratified negation or disjunction, in such a way that the stable models of this program map to the repairs of the retrieved global database.

Definition 2. Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, $\mathcal{D}$ is a source database for $\mathcal{I}$, and q is a non-recursive Datalog$^{\neg}$ query over $\mathcal{G}$. Then, a *logic specification for querying* $\mathcal{I}$ is a Datalog$^{\vee,\neg}$ program $\Pi_{\mathcal{I}}(q) = \Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_q$ such that

1. $ret(\mathcal{I}, \mathcal{D}) \rightleftharpoons \mathrm{SM}(\Pi_{\mathcal{M}}[D])$, where $\Pi_{\mathcal{M}}$ is a Datalog$^{\neg_s}$ program,
2. $rep_{\mathcal{I}}(\mathcal{D})) \rightleftharpoons \mathrm{SM}(\Pi_{\Sigma}[ret(\mathcal{I}, \mathcal{D})])$, and
3. $ans(q, \mathcal{I}, \mathcal{D}) \rightleftharpoons \{q(t) \mid q(t) \in M$ for each $M \in \mathrm{SM}((\Pi_q \cup \Pi_{\Sigma})[ret(\mathcal{I}, \mathcal{D})])\}$, where Π_q is a non-recursive Datalog$^{\neg}$ program,

where $\rightleftharpoons$ denotes a polynomial-time computable correspondence between two sets. $\square$

This definition establishes a connection between the semantics of $\Pi_{\mathcal{I}}(q)$ and the consistent answers to a query posed to $\mathcal{I}$ (Item 3) provided some syntactic transformations, which typically are simple encodings such that $\rightleftharpoons$ is a linear-time computable bijection. In particular, $\Pi_{\mathcal{I}}(q)$ is composed by three modules that can be hierarchically evaluated, i.e., $\Pi_{\mathcal{M}} \triangleright \Pi_{\Sigma} \triangleright \Pi_q$ [10], using Splitting Sets [16]:

- $\Pi_{\mathcal{M}}$ is used for retrieving data from the sources: the retrieved global database can be derived from its unique stable model (Item 1);
- Π_{Σ} is used for enforcing the constraints on the retrieved global database, whose repairs can be derived from the stable models of $\Pi_{\Sigma}[ret(\mathcal{I}, \mathcal{D})]$ (Item 2);
- finally, Π_q is used for encoding the user query q.

Our framework generalizes logic programming formalizations proposed in different integration settings, such as the ones recently proposed in [13, 4, 7, 5]. In this respect, the precise structure of the program $\Pi_{\mathcal{I}}(q)$ depends on the form of the mapping, the language adopted for specifying mapping views and user queries, and the nature of constraints expressed on the global schema. We point out that, logic programming specifications proposed in the setting of a single inconsistent database [12, 3] are also captured by our framework. Indeed, a single inconsistent database can be conceived as the retrieved global database of a GAV data integration system in which views of the mapping are assumed exact. The logic programs for consistently querying a single database are of the form $\Pi_{\mathcal{I}}(q) = \Pi_{\Sigma} \cup \Pi_q$.

4 Optimization of Query Answering

The source of complexity in evaluating the program $\Pi_{\mathcal{I}}(q)$ defined in the above section, actually lies in the conflict resolution module Π_{Σ}, and in the evaluation of Π_q. Indeed, $\Pi_{\mathcal{M}}$ is a Datalog$^{\neg_s}$ program that can be evaluated in polynomial time over the source database $\mathcal{D}$ for constructing $ret(\mathcal{I}, \mathcal{D})$, whereas Π_q is a non-recursive Datalog$^{\neg}$ program that has to be evaluated over each repair of the retrieved global database, and Π_{Σ} is in general a Datalog$^{\vee,\neg}$ program whose evaluation complexity over varying databases is at the second level of the polynomial hierarchy [12]. Furthermore, also evaluating programs with lower complexity over large data sets by means of stable models solvers, such as DLV [15] or Smodels [17], quickly becomes infeasible. This calls for suitable optimization methods speeding up the evaluation (as recently stated in [5]).

Concentrating on the most relevant and computational expensive aspects of the optimization, we focus here on Π_{Σ}, assuming that $ret(\mathcal{I}, \mathcal{D})$ is already computed, and devise intelligent techniques for the evaluation of Π_q.

Roughly speaking, in our approach we first localize in the retrieved global database $ret(\mathcal{I}, \mathcal{D})$ the facts that are not "affected" (formally specified below) by any violation. Then, we compute the repairs by taking into account only the affected facts, and finally we recombine the repairs to provide answers to the user query. Since in practice, the size of the set of the affected facts is much smaller than the size of the retrieved global database, the computation of the stable models of Π_{Σ}, i.e., repairs of $ret(\mathcal{I}, \mathcal{D})$ (Item 2 in Def. 2), over the affected facts is significantly faster than the naive evaluation of $\Pi_{\mathcal{I}}(q)$ on the whole retrieved global database.

In a nutshell, our overall optimization approach comprises the following steps:

Pruning: We first eliminate from $\Pi_{\mathcal{I}}(q)$ the rules that are not relevant for computing answers to a user query q. This can be done by means of a static syntactic analysis of the program $\Pi_{\mathcal{I}}(q)$. However, this is not a crucial aspect in our technique, and due to space limits we do not provide further details on it.

Decomposition: We localize inconsistency in the retrieved global database, and compute the set of facts that are affected by repair. Finally, we compute repairs, $\mathcal{R}_1, \ldots, \mathcal{R}_n$, of this set.

Recombination: We suitably recombine the repairs $\mathcal{R}_1, \ldots, \mathcal{R}_n$ for computing the answers to q.

In the rest of this section, we describe in detail the last two steps.

4.1 Decomposition

We start with some concepts for a single database. Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema. We call two facts $p_1(a_1, \ldots, a_n)$ and $p_2(b_1, \ldots, b_m)$, where $p_1, p_2 \in \Psi$ and each a_i, b_j is in the domain $\mathcal{U}$, *constraint-bounded in* $\mathcal{RS}$, if they occur in the same ground constraint $\sigma^g \in ground(\Sigma)$. Furthermore, for any $\sigma^g \in ground(\Sigma)$, we use $facts(\sigma^g)$ to denote the set of all facts $p(t)$, $p \in \Psi$, which occur in σ^g.

Let $\mathcal{DB}$ be a database for $\mathcal{RS}$. Then, the *conflict set* for $\mathcal{DB}$ w.r.t. $\mathcal{RS}$ is the set of facts $C_{\mathcal{DB}}^{\mathcal{RS}} = \{p(t) \mid \exists \sigma^g \in ground(\Sigma) \wedge p(t) \in facts(\sigma^g) \wedge \sigma^g \text{ is violated in } \mathcal{DB}\}$, i.e., the set of facts occurring in the constraints of $ground(\Sigma)$ that are violated in $\mathcal{DB}$.

Definition 3. Let $\mathcal{RS}{=}\langle \Psi, \Sigma \rangle$ be a relational schema and $\mathcal{DB}$ a database for $\mathcal{RS}$. Then, the *conflict closure* for $\mathcal{DB}$, denoted by ${C_{\mathcal{DB}}^{\mathcal{RS}}}^*$, is the least set such that $t \in {C_{\mathcal{DB}}^{\mathcal{RS}}}^*$ if either $t \in C_{\mathcal{DB}}^{\mathcal{RS}}$, or t is constraint-bounded in $\mathcal{RS}$ with a fact $t' \in {C_{\mathcal{DB}}^{\mathcal{RS}}}^*$. Moreover, we call $S_{\mathcal{DB}}^{\mathcal{RS}} = \mathcal{DB} - {C_{\mathcal{DB}}^{\mathcal{RS}}}^*$ and $A_{\mathcal{DB}}^{\mathcal{RS}} = \mathcal{DB} \cap {C_{\mathcal{DB}}^{\mathcal{RS}}}^*$ the *safe database* and the *affected database* for $\mathcal{DB}$, respectively. $\Box$

We drop the superscript $\mathcal{RS}$ if it is clear from the context. Intuitively, $C_{\mathcal{DB}}^*$ contains all facts involved in constraint violations, i.e., facts belonging to $C_{\mathcal{DB}}$, and facts which possibly must be changed in turn to avoid new inconsistency with Σ by repairing.

We now consider the following two subsets of all ground constraints.

(i) $\Sigma_{\mathcal{DB}}^A$ consists of all the ground constraints in which at least one fact from $C_{\mathcal{DB}}^*$ occurs, i.e., $\Sigma_{\mathcal{DB}}^A = \{\sigma^g \in ground(\Sigma) \mid facts(\sigma^g) \cap C_{\mathcal{DB}}^* \neq \emptyset\}$, and

(ii) $\Sigma_{\mathcal{DB}}^S$ consists of all the ground constraints in which at least one fact occurs which is not in $C_{\mathcal{DB}}^*$, i.e., $\Sigma_{\mathcal{DB}}^S = \{\sigma^g \in ground(\Sigma) \mid facts(\sigma^g) \not\subseteq C_{\mathcal{DB}}^*\}$.

We first show that $\Sigma_{\mathcal{DB}}^A$ and $\Sigma_{\mathcal{DB}}^S$ form a partitioning of $ground(\Sigma)$.

Proposition 1. (Separation) *Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema, and let $\mathcal{DB}$ a database for $\mathcal{RS}$. Then,*

1. $\bigcup_{\sigma^g \in \Sigma_{\mathcal{DB}}^A} facts(\sigma^g) \subseteq C_{\mathcal{DB}}^*$;

2. $\bigcup_{\sigma^g \in \Sigma^S_{\mathcal{DB}}} facts(\sigma^g) \cap C^*_{\mathcal{DB}} = \emptyset$;
3. $\Sigma^A_{\mathcal{DB}} \cap \Sigma^S_{\mathcal{DB}} = \emptyset$ *and* $\Sigma^A_{\mathcal{DB}} \cup \Sigma^S_{\mathcal{DB}} = ground(\Sigma)$.

Proof. 1. All facts occurring in a ground constraint $\sigma^g \in \Sigma^A_{\mathcal{DB}}$ must belong to $C^*_{\mathcal{DB}}$. Indeed, by definition of $\Sigma^A_{\mathcal{DB}}$, σ^g contains at least one fact t in $C^*_{\mathcal{DB}}$; each other fact in σ^g is constraint-bounded in $\mathcal{RS}$ with t, and hence it also is in $C^*_{\mathcal{DB}}$.
2. Assume by contradiction that some $\sigma^g \in \Sigma^S_{\mathcal{DB}}$ with $facts(\sigma^g) \cap C^*_{\mathcal{DB}} \neq \emptyset$ exists. Then, Definition 3 implies $facts(\sigma^g) \subseteq C^*_{\mathcal{DB}}$, which contradicts $\sigma^g \in \Sigma^S_{\mathcal{DB}}$. Part 3 is straightforward from Part 1 and Part 2. □

The separation property allows us to shed light on the structure of repairs:

Proposition 2. (Safe Database) *Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema, and let $\mathcal{DB}$ be a database for $\mathcal{RS}$. Then, for each repair $\mathcal{R} \in rep(\mathcal{DB})$ w.r.t. Σ, $S_{\mathcal{DB}} = \mathcal{R} - C^*_{\mathcal{DB}}$.*

Prior to the main result of this section, we note the following lemma:

Lemma 1. *Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema, and let $\mathcal{DB}$ be a database for $\mathcal{RS}$. Then, for each $S \subseteq S_{\mathcal{DB}}$, we have that*
1. *for each $\mathcal{R} \in rep(S \cup A_{\mathcal{DB}})$ w.r.t. Σ, $(\mathcal{R} \cap C^*_{\mathcal{DB}}) \in rep(A_{\mathcal{DB}})$ w.r.t. $\Sigma^A_{\mathcal{DB}}$;*
2. *for each $\mathcal{R}_a \in rep(A_{\mathcal{DB}})$ w.r.t. $\Sigma^A_{\mathcal{DB}}$, there exists some set of facts S', $S' \cap C^*_{\mathcal{DB}} = \emptyset$, such that $(\mathcal{R}_a \cup S') \in rep(S \cup C^*_{\mathcal{DB}})$ w.r.t. Σ.*

Armed with the above concepts and results, we now turn to the data integration settings in which we have to repair the retrieved global database $ret(\mathcal{I}, \mathcal{D})$. The following theorem shows that its repairs can be computed by looking only at $A_{ret(\mathcal{I},\mathcal{D})}$.

Theorem 1. (Main) *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Then,*
1. *$\forall \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, $\exists \mathcal{R}' \in rep(A_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. Σ such that $\mathcal{R} = \mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})} \cup S_{ret(\mathcal{I},\mathcal{D})}$;*
2. *$\forall \mathcal{R}' \in rep(A_{ret(\mathcal{I},\mathcal{D})})$, $\exists \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ w.r.t. Σ such that $\mathcal{R} = \mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})} \cup S_{ret(\mathcal{I},\mathcal{D})}$.*

Proof. Recall that $ret(\mathcal{I}, \mathcal{D}) = S_{ret(\mathcal{I},\mathcal{D})} \cup A_{ret(\mathcal{I},\mathcal{D})}$ and that $rep_{\mathcal{I}}(\mathcal{D})$ coincides with $rep(ret(\mathcal{I}, \mathcal{D}))$ w.r.t. Σ. Thus, applying Lemma 1, first Part 1 for $S = S_{ret(\mathcal{I},\mathcal{D})}$ and then Part 2 for $S = \emptyset$, we obtain that for every $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, there exists some $\mathcal{R}' \in rep(A_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. Σ of form $\mathcal{R}' = (\mathcal{R} \cap C^*_{ret(\mathcal{I},\mathcal{D})}) \cup S'$, where $S' \cap C^*_{ret(\mathcal{I},\mathcal{D})} = \emptyset$. Hence, $\mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})} = R \cap C^*_{ret(\mathcal{I},\mathcal{D})}$. By Prop. 2, every $\mathcal{R} \in rep_{\mathcal{I}}(D)$ is of form $\mathcal{R} = (\mathcal{R} \cap C^*_{ret(\mathcal{I},\mathcal{D})}) \cup S_{ret(\mathcal{I},\mathcal{D})}$. Therefore, $\mathcal{R} = (\mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})}) \cup S_{ret(\mathcal{I},\mathcal{D})}$.

Similarly, applying Lemma 1, first Part 1 for $S = \emptyset$ and then Part 2 for $S = S_{ret(\mathcal{I},\mathcal{D})}$, we obtain that for every $\mathcal{R}' \in rep(A_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. Σ, there exists some $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ w.r.t. Σ such that $\mathcal{R} = (\mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})}) \cup S'$, where $S' \cap C^*_{ret(\mathcal{I},\mathcal{D})} = \emptyset$. Moreover, Prop. 2 implies $S' = S_{ret(\mathcal{I},\mathcal{D})}$, which proves 2. □

As a consequence, for computing the stable models of the retrieved global database $rep_{\mathcal{I}}(\mathcal{D})$, it is sufficient to evaluate the program Π_{Σ} on $A_{ret(\mathcal{I},\mathcal{D})}$, i.e., to exploit the correspondence $rep_{\mathcal{I}}(\mathcal{D}) \rightleftharpoons \mathrm{SM}(\Pi_{\Sigma}[A_{ret(\mathcal{I},\mathcal{D})}])$, intersect with $C^*_{ret(\mathcal{I},\mathcal{D})}$, and unite with $S_{ret(\mathcal{I},\mathcal{D})}$. Nonetheless, computing $A_{ret(\mathcal{I},\mathcal{D})}$ is expensive in general since it requires computing the closure of $C_{ret(\mathcal{I},\mathcal{D})}$. Furthermore, in repairs of $A_{ret(\mathcal{I},\mathcal{D})}$ many facts not in $C^*_{ret(\mathcal{I},\mathcal{D})}$ might be computed which are stripped off subsequently. Fortunately, in practice, for many important cases this can be avoided: repairs can be made fully local and even focused just on the immediate conflicts in the database.

Proposition 3. *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Then,*

1. *if each $\sigma \in \Sigma$ is of the form (1) with $l > 0$, then each $\mathcal{R}'$ repair of $A_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. Σ satisfies $\mathcal{R}' \subseteq C^*_{ret(\mathcal{I},\mathcal{D})}$;*
2. *if each $\sigma \in \Sigma$ is of the form (1) with $m = 0$, then*
   i) each $\mathcal{R}'$ repair of $A_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. Σ satisfies $\mathcal{R}' \subseteq C_{ret(\mathcal{I},\mathcal{D})}$,
   ii) $C_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I}, \mathcal{D})$, and
   ii) $rep(A_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. Σ coincides with $rep(C_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. Σ. □

This proposition allows us to exploit Theorem 1 in a constructive way for many significant classes of constraints, for which it implies a bijection between the repairs of the retrieved global database, $ret(\mathcal{I}, \mathcal{D})$, and the repairs of its affected part $A_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. the constraints Σ. In particular, Condition 1 is satisfied by all constraints that do not unconditionally enforce inclusion of some fact in every repair, while Condition 2 is satisfied by constraints that can be repaired by just deleting facts from the database, such as key constraints, functional dependencies, and exclusion dependencies.

According to Theorem 1, in case of Condition 2 the set $rep_{\mathcal{I}}(\mathcal{D})$ can be obtained by simply computing the repairs of the conflicting facts, $C_{ret(\mathcal{I},\mathcal{D})}$, in place of $A_{ret(\mathcal{I},\mathcal{D})}$ and by adding $S_{ret(\mathcal{I},\mathcal{D})}$ to each repair. We also point out that the computation of the set $C_{ret(\mathcal{I},\mathcal{D})}$ can be carried out very efficiently, by means of suitable SQL statements.

The following corollary formalizes the above discussion for Condition 2.

Corollary 1. *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Assume each constraint in Σ has form (1) with $m = 0$. Then, there exists a bijection $\mu : rep_{\mathcal{I}}(\mathcal{D}) \rightarrow rep(C_{ret(\mathcal{I},\mathcal{D})})$, such that for each $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, $\mathcal{R} = \mu(\mathcal{R}) \cup S_{ret(\mathcal{I},\mathcal{D})}$ (where $C_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I}, \mathcal{D})$).* □

4.2 Recombination

Let us turn our attention to the evaluation of a user query q. According to the definition of consistent answers (Section 3.1), we need to evaluate q over each repair $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, and by Theorem 1 we can exploit the correspondence $rep_{\mathcal{I}}(\mathcal{D}) \rightleftharpoons \mathrm{SM}(\Pi_{\Sigma}[A_{ret(\mathcal{I},\mathcal{D})}])$ for computing each such $\mathcal{R}$. More precisely, we need to recombine the repairs of $A_{ret(\mathcal{I},\mathcal{D})}$ with $S_{ret(\mathcal{I},\mathcal{D})}$ computed in the decomposition step as stated by the following theorem.

Theorem 2. *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, let $\mathcal{D}$ be a source database for $\mathcal{I}$, and let q be a query over $\mathcal{G}$. Then,*

$$ans(q, \mathcal{I}, \mathcal{D}) = \bigcap_{\mathcal{R} \in rep(A_{ret(\mathcal{I},\mathcal{D})})} \Pi_q[\mathcal{R} \cap C^*_{ret(\mathcal{I},\mathcal{D})} \cup S_{ret(\mathcal{I},\mathcal{D})}] \tag{2}$$

Note that the number of repairs of $A_{ret(\mathcal{I},\mathcal{D})}$ is exponential in the number of violated constraints, and hence efficient computation of the intersection in (2) requires some intelligent strategy. Clearly, the overall approach is beneficial only if the recombination cost does not compensate the gain of repair localization. In the next section, we present an efficient technique for the recombination step.

We close this section with briefly addressing databases over infinite domains. Here, usually a safety condition is imposed on constraints and queries in order to assure finite database instances and query results. Namely, a constraint of the form (1) is safe, if each variable occurring in some B_j or ϕ_k also occurs in some A_i, and a query $q(\boldsymbol{x})$ is safe, if each variable in $\boldsymbol{x}$ is recursively bound to some value occurring in the database. Note that important types of constraints such as key, functional and exclusion dependencies are safe. It appears that under safe constraints and queries, database repairs fully localize to the *active domain*, $AD(\mathcal{DB})$, of a database $\mathcal{DB}$, i.e., the values occurring in $\mathcal{DB}$. As for repair, the (finite or infinite) domain can thus be equivalently replaced with $AD(\mathcal{DB})$, and for query answering, by $AD(\mathcal{DB})$ plus the constants in $q(\boldsymbol{x})$. Together with further domain pruning, this may lead to considerable savings.

5 A Technique for Efficient Recombination

In this section, we describe a technique for implementing the recombination step in a way which circumvents the evaluation of Π_q on each repair of $ret(\mathcal{I}, \mathcal{D})$ separately. For the sake of simplicity, we deal here with constraints of the form (1), when $l > 0$ and $m = 0$. In this case, according to Proposition 3, $rep(A_{ret(\mathcal{I},\mathcal{D})})$ coincide with $rep(C_{ret(\mathcal{I},\mathcal{D})})$, and $\mathcal{R} \subseteq C_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I}, \mathcal{D})$ for each $\mathcal{R} \in rep(C_{ret(\mathcal{I},\mathcal{D})})$. Furthermore, thesis in Theorem 2 can be rewritten as $ans(q, \mathcal{I}, \mathcal{D}) = \bigcap_{\mathcal{R} \in rep(C_{ret(\mathcal{I},\mathcal{D})})} \Pi_q[\mathcal{R} \cup S_{ret(\mathcal{I},\mathcal{D})}]$.

The basic idea of our approach is to encode all repairs into a single database over which Π_q can be evaluated by means of standard database techniques.

More precisely, for each global relation r, we construct a new relation r_m by adding an auxiliary attribute *mark*. The mark value is a string $'b_1 \ldots b_n'$ of bits $b_i \in \{0, 1\}$ such that, given any tuple t, $b_i = 1$ if and only if t belongs to the i-th repair $\mathcal{R}_i \in rep(C_{ret(\mathcal{I},\mathcal{D})}) = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, for every $i \in \{1, \ldots, n\}$ (indexing the repairs is easy, e.g. using the order in which the deductive database system computes them). The set of all marked relations constitutes a *marked* database, denoted by $M_{ret(\mathcal{I},\mathcal{D})}$. Note that the facts in $S_{ret(\mathcal{I},\mathcal{D})}$ (the bulk of data) can be marked without any preprocessing, as they belong to every repair $\mathcal{R}_i$; hence, their mark is $'11 \ldots 1'$. For our running example, the marked database derived from the repairs in Figure 2 is shown in Figure 3.

We next show how the original query q can be reformulated in a way such that its evaluation over the marked database computes the set of consistent answers to q. Before proceeding, we point out that our recombination technique also applies in the

$player_m{}^{ret(\mathcal{I}_0,\mathcal{D}_0)}$:

10	*Totti*	*RM*	$'11'$
9	*Beckham*	*MU*	$'11'$

$team_m{}^{ret(\mathcal{I}_0,\mathcal{D}_0)}$:

RM	*Roma*	10	$'10'$
MU	*Man. Utd.*	8	$'11'$
RM	*Real Madrid*	10	$'01'$

$coach_m{}^{ret(\mathcal{I}_0,\mathcal{D}_0)}$:

7	*Ferguson*	MU	$'11'$

Fig. 3. The retrieved global database of our running example after marking.

presence of constraints of general form. In such a case, the source of complexity lies in the computation of $C^*_{ret(\mathcal{I},\mathcal{D})}$.

5.1 Query Reformulation

We next focus on non-recursive Datalog queries and provide a technique for rewriting them into SQL. The extension to non-recursive Datalog$^\neg$ queries is straightforward, and is omitted due to space limits.

Let a query $q(\boldsymbol{x})$, where $\boldsymbol{x} = X_1, \ldots X_n$, be given by the rules $q(\boldsymbol{x}) \leftarrow q_j(\boldsymbol{y}_j)$, $1 \leq j \leq n$, where each $q_j(\boldsymbol{y}_j)$ is a conjunction of atoms $p_{j,1}(\boldsymbol{y}_{j,1}), ..., p_{j,m}(\boldsymbol{y}_{j,m})$, where $\boldsymbol{y}_j = \bigcup_{i=1}^{m} \boldsymbol{y}_{j,i}$. Moreover, let us call each variable Y such that $Y \in \boldsymbol{y}_{j,i}$ and $Y \in \boldsymbol{y}_{j,h}$ a *join variable of* $p_{j,i}$ *and* $p_{j,h}$.

In query reformulation, we use the following functions `ANDBIT` and `SUMBIT`:

- `ANDBIT` is a binary function that takes as its input two bit strings $'a_1 \ldots a_n'$ and $'b_1 \ldots b_n'$ and returns $'(a_1 \wedge b_1) \ldots (a_n \wedge b_n)'$, where $\wedge$ is boolean and;
- `SUMBIT` is an aggregate function such that given m strings of form $'b_{i,1} \ldots b_{i,n}'$, it returns $'(b_{1,1} \vee \ldots \vee b_{m,1}) \ldots (b_{1,n} \vee \ldots \vee b_{m,n})'$, where $\vee$ is boolean or.

Then, each $q(\boldsymbol{x}) \leftarrow q_j(\boldsymbol{y}_j)$ can be translated in the SQL statement SQL_{q_j} of the form

SELECT $\mathbf{X_1}, \ldots \mathbf{X_n}$, $(\mathbf{p_{j,1}.mark}$ ANDBIT $\ldots$ ANDBIT $\mathbf{p_{j,m}.mark})$ AS $\mathbf{mark}$
FROM $\mathbf{p_1} \ldots \mathbf{p_m}$
WHERE $\mathbf{p_{j,i}.Y{=}p_{j,h}.Y}$ (*for each join variable* $\mathbf{Y}$ *of* $\mathbf{p_{j,i}}$ *and* $\mathbf{p_{j,h}}$, $1 \leq i, h \leq m$).

In addition to the answers to $q(\boldsymbol{x}) \leftarrow q_j(\boldsymbol{y}_j)$, the statement computes the marks of the repairs in which an answer holds by applying the `ANDBIT` operator to the *mark* attributes of the joined relations. The results of all SQL_{q_j} can be collected into a view uq_m by the SQL statement SQL_{q_1} `UNION` $SQL_{q_2} \ldots$ `UNION` SQL_{q_n}. Finally, SQL_q is

SELECT $\mathbf{X_1}, \ldots \mathbf{X_n}$, SUMBIT$(\mathbf{mark})$
FROM $\mathbf{uq_m}$
GROUP BY $\mathbf{X_1}, \ldots \mathbf{X_n}$
HAVING SUMBIT$(\mathbf{mark}) =' 1 \ldots 1'$.

It computes query answers by considering only the facts that belong to all repairs.

Proposition 4. *Given a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, a source database $\mathcal{D}$ and a query q over $\mathcal{G}$, the set $ans(q, \mathcal{I}, \mathcal{D})$ coincides with the set computed by executing SQL_q over the marked database $M_{ret(\mathcal{I},\mathcal{D})}$.*

Example 5. For $q(X) \leftarrow player(X, Y, Z);\ q(X) \leftarrow team(V, W, X)$, SQL_q is

```
CREATE VIEW uqm (X, mark) AS            SELECT X, SUMBIT(mark)
SELECT Pcode, mark FROM player          FROM uqm
  UNION                                 GROUP BY X
SELECT Tleader, mark FROM team;         HAVING SUMBIT(mark) =' 11';
```

It is easy to see that the answers consist of the codes *8*, *9*, *10*. □

Since there may be exponentially many repairs in the number of violated constraints, the marking string can be of considerable length. In [9] we refine our technique to mark only the affected database part; the query is then rewritten using a split of each global relation r into a "safe" part and an "affected" part.

6 Experimental Results

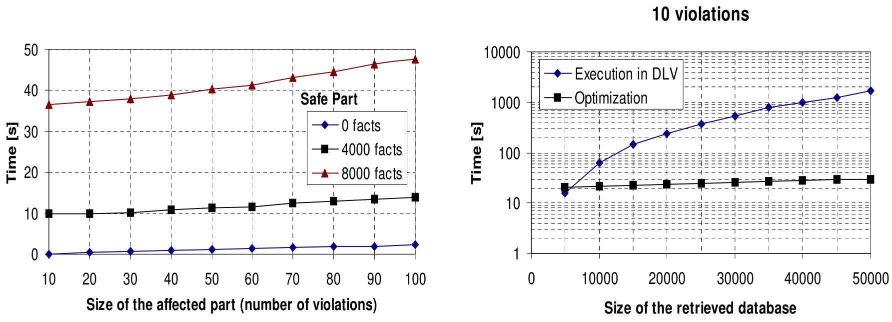
In this section, we present experimental results obtained by means of a prototype implementation that couples the DLV deductive database system [15] with Postgresql, a relational DBMS which allows for a convenient encoding of the ANDBIT and SUMBIT operators. Notice that DLV supports disjunction, which is needed for encoding universal constraints into programs Π_{Σ}, since computing consistent answers in this setting is Π_2^p-complete [12]. The experiments have been carried out on a sun4u sparc SUNW ultra-5_10 with 256MB memory and processor working at 350MHz, under Solaris SUN operating system.

Football Teams. For our running example, we built a synthetic data set $\mathcal{D}_{syn}$, in which the facts in *coach* and *team* satisfy the key constraints, while facts in *player* violate it. Each violation consists of two facts that coincide on the attribute *Pcode* but differ on the attributes *Pname* or *Pteam*; note that these facts constitute $A_{ret(\mathcal{I}_0,\mathcal{D}_{syn})}$. For the query $q(X) \leftarrow player(X, Y, Z);\ q(X) \leftarrow team(V, W, X)$, we measured the execution time of the program $\Pi_{\mathcal{I}_0}(q)$ in DLV depending on $|A_{ret(\mathcal{I}_0,\mathcal{D}_{syn})}|$ for different fixed values of $|S_{ret(\mathcal{I}_0,\mathcal{D}_{syn})}|$, viz. (i) 0, (ii) 4000, and (iii) 8000. The results are shown in Fig. 4.(a).

We point out that in Case (i), in which $\Pi_{\mathcal{I}_0}(q)$ is evaluated only on the affected part, the execution time scales well to a significant number of violations. On the other hand, the evaluation of $\Pi_{\mathcal{I}_0}(q)$ on the whole database is significantly slower; in fact, a small database of 8000 facts requires up to 40 seconds for 50 violated constraints.

Figure 4.(b) shows a comparison (in log scale) between the consistent query answering by a single DLV program and the optimization approach proposed in this paper. Interestingly, for a fixed number of violations (10 in the figure) the growth of the running time of our optimization method under varying database size is negligible. In fact, a major share ($\sim$ 20 seconds) is used for computing repairs of the affected database, plus marking and storing them in Postgresql; the time for query evaluation itself is negligible. In conclusion, for small databases (up to 5000 facts), consistent query answering by straight evaluation in DLV may be considered viable; nonetheless, for larger databases, the asymptotic behavior shows that our approach outperforms a naive implementation

Further experiments are reported in [9]. We stress that the results can be significantly improved since our implementation is not optimized. Nonetheless, its advantage over the standard technique is evident.



(a) (b)

Fig. 4. (a) Execution time in DLV system w.r.t. $|A_{ret(\mathcal{I}_0,\mathcal{D}_{syn})}|$, for different sizes of $S_{ret(\mathcal{I}_0,\mathcal{D}_{syn})}$. (b) Comparison with the optimization method.

7 Conclusion

We have described an approach to speed up the evaluation of non-monotonic logic programs modeling query answering in data integration systems. To this end, we have provided theoretical results that allow for repairing an inconsistent retrieved database by localizing the computation of repairs to its affected part. As we have shown, for important classes of constraints such as key constraints, functional dependencies, and exclusion dependencies, repairs can be restricted to the facts in the database violating the constraints, which may be only a small fragment of a large database. Furthermore, we have developed a technique for recombining such repairs to provide answers for user queries. Finally, we have experimented the viability of our approach.

We point out that our method, which is built upon repair by selection in terms of a particular preference ordering, is based on abstract properties and may be adapted to other logic programming systems as well. Furthermore, it can be generalized to other preference based repair semantics for an inconsistent database $\mathcal{DB}$. In particular, all repair semantics in which $\triangle(\mathcal{R},\mathcal{DB}) \subset \triangle(\mathcal{R}',\mathcal{DB})$, i.e., $\mathcal{R}$ is closer to database $\mathcal{DB}$ than $\mathcal{R}'$ in terms of symmetric difference, implies that $\mathcal{R}$ is preferred to $\mathcal{R}'$ can be dealt with using our method. We point out that, for instance, cardinality-based and weighted-based repair semantics satisfy this condition.

Notice that the logic formalization of LAV systems proposed in [4, 5] might be captured by our logic framework under suitable adaptations. Actually, given a source extension, several different ways of populating the global schema according to a LAV mapping may exist, and the notion of repair has to take into account a set of such global database instances. Nonetheless, analogously to our GAV framework, in [4, 5] the repairs are computed from the stable models of a suitable logic program.

On the other hand, [13, 7] address the repair problem in GAV systems with existentially quantified inclusion dependencies and key constraints on the global schema. They present techniques for suitably reformulating user queries in order to eliminate inclusion dependencies, which leads to a rewriting that can be evaluated on the global schema taking into account only key constraints. We point out that, provided such a reformulation, the logic specifications proposed in [13, 7] perfectly fit our framework.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
2. M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proc. 18th ACM Symposium on Principles of Database Systems (PODS-99)*, pp. 68–79, 1999.
3. M. Arenas, L. E. Bertossi, and J. Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Proc. 4th Int'l Conference on Flexible Query Answering Systems (FQAS 2000)*, pp. 27–41. Springer, 2000.
4. L. Bertossi, J. Chomicki, A. Cortes, and C. Gutierrez. Consistent answers from integrated data sources. In T. Andreasen *et al.*, editors, *Proc. 5th Int'l Conference on Flexible Query Answering Systems (FQAS 2002)*, LNCS 2522, pp. 71–85, 2002.
5. L. Bravo and L. Bertossi. Logic programming for consistently querying data integration systems. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 10–15, 2003.
6. A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proc. 22nd ACM Symposium on Principles of Database Systems (PODS-03)*, pp. 260–271, 2003.
7. A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 16–21, 2003.
8. J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. Technical Report `arXiv:cs.DB/0212004v1`, arXiv.org, 2002.
9. T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data integration systems. Extended Manuscript, July 2003.
10. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Trans. on Database Systems*, 22(3):364–417, 1997.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth Logic Programming Symposium*, pp. 1070–1080. MIT Press, 1988.
12. G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In P. Codognet, editor, *Proc. 17th Int'l Conference on Logic Programming (ICLP 2001)*, LNCS 2237, pp. 348–364. Springer, 2001.
13. D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *Proc. KRDB 2002*. `http://ceur-ws.org/Vol-54/`, 2002.
14. M. Lenzerini. Data integration: A theoretical perspective. In Lucian Popa, editor, *Proc. 21st ACM Symposium on Principles of Database Systems (PODS-02)*, pp. 233–246, 2002.
15. Nicola Leone et al. `DLV` homepage, since 1996. `http://www.dbai.tuwien.ac.at/proj/dlv/`.
16. V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. 11h Int'l Conference on Logic Programming (ICLP-94)*, pp. 23–38, 1994. MIT-Press.
17. Ilkka Niemelä et al. Smodels homepage, since 1999. `http://www.tcs.hut.fi/Software/smodels/`.
18. T. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.
19. D. Seipel. DisLog - a disjunctive deductive database prototype (system description). In F. Bry, B. Freitag, and D. Seipel, editors, *Proc. 12th Workshop on Logic Programming (WLP '97)*. LMU München, September 1997.
20. David S. Warren et al. `XSB` homepage, since 1997. `http://xsb.sourceforge.net/`.