I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG WISSENSBASIERTE SYSTEME

OPTIMIZATION METHODS FOR LOGIC-BASED QUERY ANSWERING FROM INCONSISTENT DATA INTEGRATION SYSTEMS

Thomas Eiter Michael Fink Gianluigi Greco
Domenico Lembo

INFSYS RESEARCH REPORT 1843-05-05

JULY 2005

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 1843-05-05, JULY 2005

OPTIMIZATION METHODS FOR LOGIC-BASED QUERY ANSWERING FROM INCONSISTENT DATA INTEGRATION SYSTEMS

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

Abstract. Information integration systems providing the user with transparent access to heterogeneous data sources through a unified global view of all data, have emerged as a crucial issue in many application domains. This global view usually comprises integrity constraints which should be satisfied by the data retrieved from the sources. However, they might be often violated, and suitable approaches for handling such a situation are needed, taken inconsistency and incompleteness of source data into account. To this end, several recent works reduce query answering in data integration systems to non-monotonic logic programming. This branch of computational logic is well-suited for dealing with inconsistent and incomplete information, and offers the expressiveness which is needed to model query answering, which is in the worst case intractable. Furthermore, advanced implementations of systems including stable model engines like DLV and Smodels for evaluation of non-monotonic logic programs are available. Nonetheless, a naive reduction of data integration to non-monotonic logic programs is infeasible for larger data sets, and calls for optimizations. We address this issue and present several techniques of different nature which make a non-monotonic logic programming approach effective. The first is a technique to prune a logic program for query answering such that only the relevant rules are kept. The second is a sophisticated technique, developed at a generic level, which aims at localizing inconsistency and "repairing" it in a decomposition manner. Finally, we present a technique to combine non-monotonic logic programming and commercial relational database engines in order to facilitate efficient query answering from repairs computed for an inconsistent global data view. Our methods are not bound to a specific non-monotonic logic programming system. In particular, the results on localization and decomposition are independent of a logic program approach, and may be exploited for efficient realization of advanced data integration systems in general.

Keywords: advanced data integration, database repairs, inconsistency management in databases, computational logic, non-monotonic logic programs, stable models.

---

[1]Institute of Information Systems, Knowledge-Based Systems Group, TU Vienna, Favoritenstraße 9-11, A-1040 Vienna, Austria. E-mail: (eiter|michael)@kr.tuwien.ac.at.

[2]Mathematics Dept., Univ. Calabria, Via Pietro Bucci 30B, I-87036 Rende, Italy. E-mail: ggreco@mat.unical.it.

[3]DIS University of Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy. E-mail: lembo@dis.uniroma1.it.

Acknowledgements: This work has been partially supported by the European Commission FET Programme Projects IST-2002-33570 INFOMIX and IST-2001-37004 WASP, and the Austrian Science Funds (FWF) project P18019-N04.

A conference version of this paper, containing some of the results in preliminary form, appeared in: Proceedings 19th Int'l Conference on Logic Programming (ICLP 2003), Mumbai, India, Dec. 2003 [25].

Contents

1 Introduction

Recent developments in IT such as the expansion of the Internet and the World Wide Web, have made available a huge number of information sources, generally autonomous, heterogeneous and widely distributed. Therefore, information integration has become a crucial challenge at the current evolutionary stage of IT infrastructures. A data integration system aims at providing transparent access to data dispersed over many heterogeneous information sources, and relieving the users from the burden of having to identify those sources that store data relevant for their queries, 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 a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ is the *global schema*, which specifies the global elements exported to users, $\mathcal{S}$ is the *source schema*, which describes the structure of the data sources in the system, and $\mathcal{M}$ is the *mapping*, which establishes the relationship between the sources and the global schema. There are basically two approaches for specifying the mapping [46]: the *Global-As-View (GAV)* approach, which requires that a view over the sources is associated with every element of the global schema, so that its the meaning is specified in terms of the data residing at the sources, and the *Local-As-View* (LAV) approach, which conversely requires the sources to be defined as views over the global schema. In this paper, we focus on the GAV approach, which is generally considered sufficiently simple and effective for practical purposes. Notice, however, that optimization techniques for query answering that we propose in the following apply also to existing approach to consistent query answering in LAV (see e.g., [3, 14]), under simple adaptations.

Usually, the global schema $\mathcal{G}$ is assumed to be relational, generally enriched with integrity constraints, denoted with Σ, issued on relation symbols. Views in the mapping $\mathcal{M}$ are specified over the (relational) source schema $\mathcal{S}$ in a language which often amounts to a fragment of Datalog (possibly extended with stratified negation).

Example 1.1 As a running example, we consider a data integration system $\mathcal{I}_0 = \langle \mathcal{G}_0, \mathcal{S}_0, \mathcal{M}_0 \rangle$, which provides information about soccer teams of the 2002/03 edition of the U.E.F.A. Champions League. 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 specify that the keys of $player$, $team$, and $coach$, are the sets of attributes $\{Pcode, Pteam\}$, $\{Tcode\}$, and $\{Ccode, Cteam\}$, respectively, and that a coach can neither be a player nor a team leader. The source schema $\mathcal{S}_0$ comprises the relation symbols s_1, s_2, s_3 and s_4. Finally, the mapping $\mathcal{M}_0$ is defined by the Datalog program

$player(x, y, z) \leftarrow s_1(x, y, z, w)$;

$team(x, y, z) \leftarrow s_2(x, y, z)$;

$team(x, y, z) \leftarrow s_3(x, y, z)$;

$coach(x, y, z) \leftarrow s_4(x, y, z)$,

which casts into rules the mapping views. In the program, we used global relation symbols in the left-hand side (head) of the rules to explicitly trace the correspondence between global elements and the associated views in the mapping. □

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

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

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

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

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

7	*Camacho*	RM

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

Since in general integrated sources are originally autonomous, their data, filtered through the mapping, are likely not to satisfy the constraints on the global schema. This can be easily seen if we explicitly construct a global database instance by retrieving data from the sources according to the mapping specification.

Example 1.2 Assuming that the information in the sources is given by the database $\mathcal{D}_0 = \{\ s_1(10, \textit{Totti}, \textit{RM}, 27),\ s_1(9, \textit{Beckham}, \textit{MU}, 28),\ s_2(RM, \textit{Roma}, 10),\ s_3(MU, \textit{Man. Utd.}, 8),\ s_3(RM, \textit{Real Madrid}, 10),\ s_4(7, \textit{Camacho}, RM)\}$, 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 $team(RM, \textit{Roma}, 10)$ and $team(RM, \textit{Real Madrid}, 10)$, which coincide on $Tcode$ but differ on $Tname$. □

Therefore, when the user issues a query Q on the global schema (which is generally expressed in a fragment of non-recursive Datalog with negation), the problem arises of establishing which are the answers that have to be returned to Q from an inconsistent data integration system.

The standard approach to remedy to this problem is through data cleaning [13]. This approach is procedural in nature, and is based on domain-specific transformation mechanisms applied to the data retrieved from the sources. An alternative declarative approach has been proposed in the last years in the field of *consistent query answering*, which studies the definition (and computation) of informative answers to queries posed to inconsistent databases (see, e.g., [2, 36, 16]). The proposals developed in this field rely on the notion of repair as introduced in [2]: a repair of a database is a new database that satisfies the constraints in the schema, and minimally differs from the original one. 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. For a survey on query answering over inconsistent databases, see [11]. The above results are not specifically tailored to the case of different consistent sources that are mutually inconsistent, that is the case of interest in data integration. More recently, some papers (see, e.g., [12, 17, 14]) have tackled data inconsistency in a data integration setting, where the basic idea is to apply the repairs to data retrieved from the sources, again according to some minimality criteria.

In the last years, several approaches to formalize repair semantics by using logic programs have been proposed both for single inconsistent databases and for data integration systems, cf. [4, 8, 12, 14, 15, 17, 36]. As for the data integration setting, the idea common to these works is to encode the constraints Σ of the global schema $\mathcal{G}$ into a logic program, Π, using unstratified negation and/or disjunction (i.e., by a Datalog$^{\vee,\neg}$ program), such that the stable models of this program yield the repairs of the global database. Answering a user query, then amounts to cautious reasoning over the logic program Π augmented with the query, cast into rules, and the facts retrieved from the sources.

This approach has some attractive features. An important one is that Datalog$^{\vee,\neg}$ programs serve as *executable logical specifications of repair*, and thus provide a language for expressing repair policies in a fully declarative manner rather than in a procedural way. Furthermore, *reasoning about specifications*, their properties and behavior, is much better facilitated than for procedural repair specifications, since reasoning

about programs is one of the principal issues in logic programming, and has been abundantly studied. Finally, extensions to the Datalog$^{\vee,\neg}$ language which allow e.g. to handle priorities and weight constraints (cf. [48, 62]), provide a useful set of constructs for expressing also more involved criteria that repairs should satisfy, which possibly have to be customized to a particular application scenario. Here, logic programming specifications may serve as a useful test-bed for development, since variants of repair can be quickly realized and experimented with.

A major drawback of the logic programming specification approach is that with current (yet still improving) implementations of stable model engines, which are needed for program evaluation, such as DLV [48] or Smodels [62], 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 [14].

1.1 Efficient Evaluation of Logic Programs

Datalog$^{\vee,\neg}$ programs [26] have been introduced as a tool for knowledge representation and commonsense reasoning, with the aim of modeling incomplete data [54, 7]. In the past years, several alternative semantics for these programs have been proposed in the literature (see [23, 55] for comprehensive surveys). Currently, the most widely accepted semantics is the *stable model semantics* proposed by Gelfond and Lifschitz [35], which is, in fact, the one exploited in the data integration setting.

According to this semantics, Datalog$^{\vee,\neg}$ programs may have several alternative models and capture the complexity class Σ_2^P [26], i.e., they allow us to express, in a precise mathematical sense, *every* generic class of finite relational structures that is decidable in nondeterministic polynomial time with an oracle in NP. However, the expressiveness comes at the price of a higher computational cost in the worst case: The principal reasoning tasks, brave reasoning and cautious reasoning, for disjunctive logic programs are Σ_2^P-complete and Π_2^P- complete, respectively [26]; in absence of disjunction, they are NP-complete and co-NP-complete, respectively (cf. [22]).

In the 1990s, Datalog$^{\vee,\neg}$ received little attention for practical application in the database community, mainly because of its worst case complexity. However, the recent emergence of advanced data integration scenarios which strictly demand co-NP and often even Π_2^P data complexity (see, e.g., [17, 18, 36]), has triggered interest in this language and, specifically, in the design of efficient techniques for stable model computation and query answering, extending and complementing previous and ongoing research on optimizing disjunctive Datalog engines such as DLV. In this line of research, three main streams can be identified.

• *Binding propagation techniques:* The goal of these techniques is to use the constants appearing in the query to reduce the size of the instantiation by eliminating "a priori" a number of ground instances of the rules which cannot contribute to the derivation of the query goal. The key idea is to materialize, by suitable *adornments*, binding information for IDB predicates which would be propagated during a top-down computation. These are strings of the letters b and f, denoting bound or free for each argument of an IDB predicate. Due to its efficiency and its generality, the *magic set method* [6, 9] is the most-well known method adhering to such an approach. Other focusing methods, e.g. supplementary magic sets and other special techniques for linear and chain queries, have been proposed as well (see, e.g., [38, 64, 57]). Numerous extensions and refinements of the magic set method exist, addressing e.g. query constraints [63], modular stratification and well-founded semantics [58, 42], or integration into cost-based query optimization [60], and the research on further extensions is still active. For instance, [10] and [37, 21] present magic sets techniques for *soft-stratifiable* programs, and for disjunctive programs, respectively.

• *Equivalence of logic programs and optimization:* Here, the objective is to "optimize" a given program by means of local simplification rules, in the spirit of conjunctive query minimization in databases [64]. Typically, redundant literals are removed and rule bodies simplified (assuming that this speeds up rule evaluation), as well as redundant and unproductive rules pruned from the program. To this end, basic issues like semantic equivalence of programs arise here. It is well-known in the database theory that already equivalence of plain Horn datalog programs is undecidable [61], and the major problems for static optimization of Datalog$^{\vee,\neg}$ queries are decidable only for restricted fragments, cf. [40]. Recent research in non-monotonic logic programming considers refined notions of equivalence such as uniform equivalence [59] and strong equivalence [49], which are decidable for plain Horn datalog programs [59] and arbitrary Datalog$^{\vee,\neg}$ programs [51]. These notions of equivalence are considered to be a useful tool for more general optimizations than local rule simplification and pruning, such as replacing program patterns with equivalent (optimized) rule sets from a library.

• *Heuristics:* Recent research is devoted to heuristics for guiding the search for a model, inspired by preliminary similar work in the SAT community, cf. [28, 29]. For non-monotonic logic programs this issue is more complicated, however, since the semantics of rules is more involved (intuitively, by minimality constraints). As a matter of fact, different heuristics may result in quite different performance for model finding on benchmark sets. Improving heuristics is a challenging issue and active research area for increasing the performance of stable model engines.

Even though the techniques above are quite effective for the optimization of logic programs modelling reasoning tasks mainly in AI applications, they do not warrant in general sufficient efficiency and scalability for the usage of Datalog$^{\vee,\neg}$ programs under the stable model semantics in real data integration scenarios.

Indeed, such techniques do not benefit of the peculiarities of the specific application domain, e.g., they do not exploit the semantics of the logic specification to be optimized and the fact that each stable model is a way of repairing a database instance, and do not seriously face the problems emerging in this setting, where the actual bottleneck is the (huge) size of the source databases on which the stable models engines have to reason for computing answers to user queries.

Moreover, these techniques have been designed for stable models engines which at the present time result impractical for real database applications. Most of them currently miss primitives for interfacing with database management systems (which is needed for efficiently retrieving relevant data), as well as mechanisms for directly handling answering non-ground queries (which then must be simulated by time-consuming multiple ground query evaluations). Only very recently, such features are available in DLV, as a result of research activities carried out in line with the present work. However, the scalability problems are not completely solved in stable model engines by these improvements.

In order to tackle this issue, we study the problem of efficient evaluation of logic program specifications for querying data integration systems. We introduce practical techniques, which are specifically tailored to optimize logic programs for querying data integration systems, thereby devising an approach which is orthogonal to most of the optimization techniques discussed above. Our main idea is to "localize" the inconsistency in the database to be repaired, and limit the inefficient computation in the stable model engine to a very small fragment of the input, obtaining fast query-answering, even in a powerful data-integration framework. Importantly, the techniques are designed in a way that can be easily accommodated in the architecture of a data integration prototype, whose development has been carried out within the EU project "INFOMIX: Boosting Information Integration" (IST-2001-33570) [47].

1.2 Contributions

The main contributions of this paper are briefly summarized as follows.

(1) We present a basic formal model of data integration via logic programming specification, which abstracts from several proposals in the literature [4, 8, 12, 14, 17, 36]. Results which are obtained on this model may then be inherited by the respective approaches. In this model, the different components of a logic programming specification, viz. a component for mapping the data at the source to the global relation, an "integration" component which reconciles inconsistency and deals with incompleteness, as well as a component for evaluating the query are explicitly articulated. These components are naturally organized in a hierarchical manner, which can be profitably exploited by a logic programming engine for evaluation of a logic programming specification of data integration and query answering.

(2) In order to reduce the complexity of naive query evaluation, we foster a *repair localization approach*, in which irrelevant rules are discarded and the retrieved data is decomposed into two parts: facts which will possibly be touched by a repair, called the "affected database," and facts which for sure will be not, called the "safe database." The idea at the heart of this approach is to reduce the usage of the non-monotonic logic program to the essential part for conflict resolution. This requires that some technical conditions are met in order to make the part "affected" by a repair small (ideally, as small as possible). We develop this localization approach at a generic database level, independent of a commitment to a particular definition of repairs and query answering program, but based on a common setting of repair semantics: the repairs of the (retrieved) database are characterized by the minimal (non-preferred) databases from a space of candidate repairs with a preference order. Examples from the literature are set-inclusion based orderings [32, 2, 4, 8, 14, 16, 17, 18, 36], cardinality-based [4, 53] ordering, and weighted-based ordering [52]. More precisely, we establish localization results for three increasingly expressive classes of constraints (where each constraint must involve at least one database relation):

- The first class contains all constraints of the form $\forall\vec{x}\alpha(\vec{x}) \supset \phi(\vec{x})$, where $\alpha(\vec{x})$ is a nonempty conjunction of atoms on database relations and ϕ is a disjunction of built-in literals. These constraints are semantically equivalent to denial constraints [19].
- The second class allows more general constraints of form $\forall\vec{x}\alpha(\vec{x}) \supset \beta(\vec{x}) \vee \phi(\vec{x})$, where $\alpha(\vec{x})$ and $\phi(\vec{x})$ are as above and $\beta(\vec{x})$ is a disjunction of atoms on database relations.
- The third class are all universal constraints in clausal form, i.e., of the form above where either $\alpha(\vec{x})$ or $\beta(\vec{x})$ is nonempty. Thus, semantically, this class captures all universal constraints.

Furthermore, we show that under mild conditions, repairs can be factorized into independent components, which can be processed orthogonally to each other.

(3) We develop *techniques for recombining* the decomposed parts for query answering, which interleave a logic programming and a relational database engine. 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. Hence, in our overall approach, the attractive features of a non-monotonic logic programming system can be fruitfully combined with the strengths of an efficient relational database engine. This is substantiated with an experimental prototype implementation in which the stable model engine DLV is coupled with the DBMS PostgreSQL.

(4) We present a general architecture for a data integration system implementing the optimization techniques. The architecture demonstrates that it is possible to implement localization approaches profitably on top of an answer set engine. This entails a significant speed up for query answering in advanced data integration systems.

(5) Finally, we thoroughly assess the effectiveness of our approach in a suite of experiments, which involve both synthetic and real data. The experimental results are encouraging and show that the system scales well even in real-world scenarios.

Our results on localization extend and generalize previous localization results which have been utilized (sometimes tacitly) for particular repair orderings and classes of constraints, e.g., for denial constraints and repairs which are closest to the original database measured by symmetric difference [19].

We point out that our overall contribution is twofold in nature. One the one hand it is foundational. Our results on inconsistency management provide a unifying logic-based view of previous approaches to query answering from inconsistent data integration systems, and shed light on the interaction of integrity constraints and the structure of preference-based repairs. These results can be exploited for efficient implementation of data integration systems in general, independent of a logic-based approach. Furthermore, they may be fruitfully applied in the context of data exchange [30], which fosters materialization of source data into a target global schema, enriched with integrity constraints. On the other hand of our contribution is practical as well, aiming at an efficient implementation of the theoretical results. To this end, we have developed innovative methods and techniques relying on existing technologies offered by stable models and relational engines. Importantly, these techniques are not bound to a particular stable models or relational engine, and may be customized to any of the available established systems.

1.3 Organization

The remainder of this paper is organized as follows. Section 2 contains preliminaries on Datalog$^{\vee,\neg}$ programs and introduces the notation for the relational data model used throughout the paper. In Section 3, we present our generic framework for data integration and query answering, followed in Section 4 by an emerging logic programming framework for consistent query answering. There, also several examples of logic-programming based approaches are considered. Section 5 then presents our overall optimization approach. Specifically, it discusses three distinct steps:

(i) a *pruning* phase, in which we eliminate from the logic specification the rules that are not relevant for computing answers to a user query Q;

(ii) a *decomposition* phase in which we localize inconsistency in the database $\mathcal{DB}$ to be repaired, compute the set of facts in $\mathcal{DB}$ that are affected by repair (denoted $\mathcal{A}_{\mathcal{DB}}$), and obtain the repairs of $\mathcal{DB}$ by actually repairing the (smaller) database $\mathcal{A}_{\mathcal{DB}}$; and

(iii) a *recombination* phase in which we suitably recombine the repairs for computing the answers to Q.

Since the overall approach is beneficial only if the recombination cost does not compensate the gain of repair localization, we present in Section 6 some effective recombination techniques which interleave a stable models engine and a relational DBMS. The effectiveness of the approach is assessed in Section 7 by experiments with both synthetic and real data. Finally, in Section 8 we draw our conclusions.

2 Preliminaries

In this section, we recall some concepts and notation that will be useful throughout this paper. In particular, we describe the query language which we are interested in, and present the basic notions of the relational model, on which we will construct our formal framework for modeling query answering in data integration systems. For an exhaustive treatment and further background, we refer the reader to [1, 26, 35].

2.1 Disjunctive Datalog with Negation

A *Datalog*$^{\vee,\neg}$ rule ρ is an expression of the form

$$s_1(\vec{v}_1) \vee \ldots \vee s_n(\vec{v}_n) \leftarrow r_1(\vec{u}_1), \ldots, \ r_k(\vec{u}_k), \ not\ r_{k+1}(\vec{u}_{k+1}), \ldots, \ not\ r_{k+m}(\vec{u}_{k+m}) \tag{1}$$

where $s_i(\vec{v}_i)$, $r_j(\vec{u}_j)$ are atoms in a relational first-order language $\mathcal{L}$, i.e., s_i resp. r_j are relation (predicate) names and $\vec{v}_i = v_{i,1}, \ldots, v_{i,n_i}$ resp. $\vec{u}_j = u_{j,1}, \ldots, u_{j,n_j}$ are lists of variables and constants (*terms*) matching the arity of s_i resp. r_j. Here, "not" is *negation as failure* (or *default negation*) and "," is conjunction. The disjunction left of "$\leftarrow$" is the *head* of the rule, denoted $head(\rho)$, while the conjunction right of "$\leftarrow$" is its *body*, denoted $body(\rho)$. Unless stated otherwise, we require that ρ is *safe*, i.e., each variable occurring in ρ occurs in some positive body atom $r_i(\vec{u}_i)$, $1 \leq i \leq k$, where r_i is different from a built-in relation. Such relations, if present, may occur only in rule bodies.

A *Datalog*$^{\vee,\neg}$ *program* $\mathcal{P}$ is a finite set of Datalog$^{\vee,\neg}$ rules. Important syntactic fragments of Datalog$^{\vee,\neg}$ with decreasing expressiveness are the class of *normal programs*, denoted Datalog$^{\neg}$, in which $n = 1$ for all rules, and the class of *stratified normal* programs, denoted Datalog$^{\neg s}$, and of *non-recursive* programs, which are defined as follows. Each Datalog$^{\neg}$ program $\mathcal{P}$ has an associated *dependency graph*, which is a directed graph $G(\mathcal{P}) = \langle V, E \rangle$, whose vertices V are the predicates occurring in $\mathcal{P}$ and where E contains an arc $r \rightarrow s$ if r and s occur in the head respectively body of some rule ρ in $\mathcal{P}$. If in some such ρ, s occurs under negation, the arc is labeled with '$*$.' Then, program $\mathcal{P}$ is *stratified*, if $G(\mathcal{P})$ has no cycle with an arc labeled '$*$,' and *non-recursive*, if $G(\mathcal{P})$ is acyclic. Notice that a non-recursive Datalog program can be rewritten in terms of a *Union of Conjunctive Queries*, i.e., a set of rules of the form 1 where $n = 1$ and $m = 0$, all having the same head predicate.

If a rule ρ of form (1) has $k = m = 0$, then ρ is a *fact* and "$\leftarrow$" is omitted. For any Datalog$^{\vee,\neg}$ program $\mathcal{P}$ and set of facts $\mathcal{D}$, we denote by $\mathcal{P}[\mathcal{D}]$ the program $\mathcal{P} \cup \mathcal{D}$.

Predicate names in $\mathcal{P}$ are called *extensional* (*EDB predicates*) if they occur only in the body of the rules in $\mathcal{P}$, otherwise they are called *intensional* (*IDB predicates*). In other words, IDB predicates are defined by the rules of the program, whereas EDB predicates are defined by the facts of a database (see Section 2.2).

The semantics of a program $\mathcal{P}$ is defined in terms of its Herbrand instantiation (or grounding) $ground(\mathcal{P})$ with respect to the language $\mathcal{L}$ (often, the language generated by $\mathcal{P}$), which consists of all instances of rules ρ of form (1) in $\mathcal{P}$ in which variables are replaced with constant symbols from $\mathcal{L}$.

Let $\mathcal{B}_\mathcal{L}$ be the set of all ground atoms constructible from the predicate and constant symbols in $\mathcal{L}$. An *interpretation for* $\mathcal{P}$ is any subset $I \subseteq \mathcal{B}_\mathcal{L}$. A ground (variable-free) atom $p(\vec{c})$ is true in I, if $p(\vec{c}) \in I$, and false in I otherwise; a ground rule ρ of form (1) is *satisfied* by I, if either some $s_i(\vec{v}_i)$ or $r_{k+j}(\vec{u}_{k+j})$, $0 < j \leq m$, is true in I, or some $r_j(\vec{u}_j)$, $1 \leq j \leq k$, is false in I. The interpretation I is a model of $\mathcal{P}$, if I satisfies all rules in $ground(\mathcal{P})$.

The *stable model semantics* [35], which we adopt here, assigns to every "not"-free Datalog$^{\vee,\neg}$ program $\mathcal{P}$ the set $\mathrm{MM}(\mathcal{P})$ of its minimal models, where a model M of $\mathcal{P}$ is minimal, if no proper subset of M is a model of $\mathcal{P}$. To programs $\mathcal{P}$ with negation, it assigns every interpretation I such that I is a minimal model of the *Gelfond-Lifschitz reduct* $\mathcal{P}$ w.r.t. I, which is the "not"-free program obtained from $ground(\mathcal{P})$ by

- deleting each rule having a ground literal $not\ p(\vec{c})$ in the body, such that $p(\vec{c}) \in I$;
- deleting the negative body from the remaining rules.

Any such I is called a *stable model* of $\mathcal{P}$; the set of stable models of $\mathcal{P}$ is denoted by $\mathrm{SM}(\mathcal{P})$. Note that for "not"-free programs, minimal models and stable models coincide, and that positive disjunction-free (resp. stratified) programs have a unique stable model [35].

2.2 Data Model

We assume a countable infinite database domain $\mathcal{U}$ whose elements can be referenced by constants $c_1, c_2, \ldots$ under the *unique name assumption*, i.e., different constants denote different real-world objects. A *relational schema* (or simply *schema*) $\mathcal{RS}$ is a pair $\langle \Psi, \Sigma \rangle$, where:

- Ψ is a finite set of relation (predicate) symbols, each with an associated positive arity.
- Σ is a finite set of *integrity constraints* (ICs) expressed on the relation symbols in Ψ, i.e., assertions that are intended to be satisfied by database instances. We consider here universally quantified constraints [1], more specifically first-order sentences of the form

$$\forall \vec{x}\, A_1(\vec{x}_1) \wedge \cdots \wedge A_l(\vec{x}_l) \supset B_1(\vec{y}_1) \vee \cdots \vee B_m(\vec{y}_m) \vee \phi_1(\vec{z}_1) \vee \cdots \vee \phi_n(\vec{z}_n), \tag{2}$$

  where $l + m > 0$, $n \geq 0$, the $A_i(\vec{x}_i)$ and the $B_j(\vec{y}_j)$ are atoms on Ψ, the $\phi_k(\vec{z}_k)$ are atoms or negated atoms over built-in relations such as equality $=$, inequality $\neq$ etc. (if available), $\vec{x}$ is a list of all variables occurring in the formula, and the $\vec{x}_i$, $\vec{y}_j$, and $\vec{z}_k$ are lists of variables and constants.[1] The conjunction left (resp., disjunction right) of "$\supset$" is the *body* (resp. *head*) of the constraint. When the body is an empty conjunction, we omit the "$\supset$" symbol and simply write the head of the constraints.

Notice that (2) is a clausal normal form, in which arbitrary universal constraints on a relational schema can be expressed. Among them are many classical constraints, such as

- functional dependencies $\forall \vec{x} y_1 \vec{z}_1 y_2 \vec{z}_2\, p(\vec{x}, y_1, \vec{z}_1) \wedge p(\vec{x}, y_2, \vec{z}_2) \supset y_1{=}y_2$;
- key constraints $\forall \vec{x}\vec{y}\vec{z}\, p(\vec{x}, \vec{y}) \wedge p(\vec{x}, \vec{z}) \supset y_1{=}z_1, \ldots, y_n{=}z_n$, where $\vec{y} = y_1, \ldots, y_n$ and $\vec{z} = z_1, \ldots, z_n$;
- exclusion dependencies $\forall \vec{x}\vec{y}\vec{z}\, p_1(\vec{x}, \vec{y}) \wedge p_2(\vec{x}, \vec{z}) \supset \bot$, where $\bot$ is the empty disjunction;
- denial constraints $\forall \vec{x}_1 \cdots \vec{x}_n \neg(P_1(\vec{x}_1) \wedge \cdots P_n(\vec{x}_n) \wedge \alpha(\vec{x}_1, \ldots, \vec{x}_n))$ where $\alpha(\vec{x}_1, \ldots, \vec{x}_n)$ is a Boolean combination of built-in atoms [19]; and,
- inclusion dependencies of the form $\forall \vec{x}\, p_1(\vec{x}) \supset p_2(\vec{x})$.

[1] The condition $l + m > 0$ excludes constraints involving only built-in relations, which are irrelevant from a schema modeling perspective.

Example 2.1 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:

$$\begin{aligned}
&\forall x, y, y', z \; player(x, y, z) \wedge player(x, y', z) \supset y{=}y', \\
\forall x, y, y', z, z' \; &team(x, y, z) \wedge team(x, y', z') \supset y{=}y', \\
\forall x, y, y', z, z' \; &team(x, y, z) \wedge team(x, y', z') \supset z{=}z', \\
&\forall x, y, y', z \; coach(x, y, z) \wedge coach(x, y', z) \supset y{=}y', \\
&\forall x, y, y', z \; coach(x, y, z) \wedge player(x, y', z) \supset \bot, \\
&\forall x, y, y', z \; coach(x, y, z) \wedge team(z, y', x) \supset \bot.
\end{aligned}$$

The first four rows encode the key constraints, whereas the last two rows model the two constraints stating that, for any given team, a coach cannot be a player or a team leader (exclusion dependencies). □

In our localization approach, we pay special attention to the following subclasses of constraints:

- Constraints with only built-in relations in the head (i.e., $m = 0$ in (2)). The class of these constraints, which we denote by $\mathbf{C}_0$, is a clausal normal form of denial constraints. This class (semantically) includes key constraints, functional dependencies, and exclusion dependencies, for instance.
- Constraints with non-empty body (i.e., $l > 0$ in 2)). We denote the class of these constraints, which permit conditional generation of tuples in the database, by $\mathbf{C}_1$.

Note that $\mathbf{C}_0 \subseteq \mathbf{C}_1$ (since $l + m > 0$). We next define the semantics of a database scheme.

For any set of relation symbols Ψ as above, let $\mathcal{F}(\Psi)$ denote the set of all facts $r(t)$, where $r \in \Psi$ has arity n and $t = (c_1, \ldots, c_n) \in \mathcal{U}^n$ is an n-tuple of constants from $\mathcal{U}$. A *database instance* (or simply *database*) for Ψ is any finite $\mathcal{DB} \subseteq \mathcal{F}(\Psi)$. For any relation r, we denote by $r^{\mathcal{DB}}$ its extension in $\mathcal{DB}$, which is the set of tuples $\{t \mid r(t) \in \mathcal{DB}\}$. Furthermore, we denote by $\mathcal{DB}(\Psi)$ the set of all databases for Ψ.

For any relation schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$, in abuse of notation $\mathcal{F}(\mathcal{RS})$ and $\mathcal{DB}(\mathcal{RS})$ denote $\mathcal{F}(\Psi)$ and $\mathcal{DB}(\Psi)$, respectively. A *database* for $\mathcal{RS}$ is a database for Ψ.

A constraint σ^g is *ground*, if it is variable-free. For any ground constraint σ^g, we denote by $facts(\sigma^g)$ the set of all facts $p(t) \in \mathcal{F}(\mathcal{RS})$ which occur in σ^g, and for any set Σ^g of ground constraints we denote $facts(\Sigma^g) = \bigcup_{\sigma^g \in \Sigma^g} facts(\sigma^g)$. For any constraint $\sigma = \forall \vec{x} \, \alpha(\vec{x})$, we denote by $ground(\sigma)$ the set of its *ground instances* $\theta(\alpha(\vec{x}))$, where θ is any substitution of the variables $\vec{x}$ by constants from $\mathcal{U}$, and for any set of constraints Σ, $ground(\Sigma) = \bigcup_{\sigma \in \Sigma} ground(\sigma)$.

Given $\mathcal{R} \subseteq \mathcal{F}(\Psi)$, where $\Psi = \{r_1, \ldots, r_n\}$, $\mathcal{R}$ *satisfies* a constraint σ, denoted $\mathcal{R} \models \sigma$, if σ is true on the relational structure $(\mathcal{U}, r_1^{\mathcal{R}}, \ldots, r_n^{\mathcal{R}}, c_1^{\mathcal{R}}, c_2^{\mathcal{R}}, \ldots)$ where $c_i^{\mathcal{R}} = c_i$, for all $c_i \in \mathcal{U}$ (that is, each $\sigma' \in ground(\sigma)$ evaluates to true), and *violates* σ otherwise; $\mathcal{R}$ *satisfies* a set of constraints Σ (or, *is consistent with* Σ), denoted $\mathcal{R} \models \Sigma$, if $\mathcal{R} \models \sigma$ for every $\sigma \in \Sigma$, and *violates* Σ otherwise. A database schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ is *consistent*, if there exists some database $\mathcal{DB}$ for $\mathcal{RS}$ consistent with Σ.

Finally, a *query over* $\mathcal{RS}$ is a mapping which assigns to each database $\mathcal{D}$ for $\mathcal{RS}$ a set of n-tuples over $\mathcal{U}$, where n is fixed. We consider here queries naturally expressed using Datalog$^{\vee,\neg}$ programs, and refer to [1] for further background about queries.

Formally, a *Datalog$^{\vee,\neg}$ query* Q (over $\mathcal{RS}$) is a pair $\langle q, \mathcal{P} \rangle$ where $\mathcal{P}$ is a safe Datalog$^{\vee,\neg}$ program such that relation symbols from Ψ occur only as EDB predicates of $\mathcal{P}$, and q is an IDB predicate of $\mathcal{P}$. The *arity of* Q is the arity of q. Given any database $\mathcal{D}$ for $\mathcal{RS}$, the *evaluation of* Q *over* $\mathcal{D}$, denoted $Q[\mathcal{D}]$, is

$$Q[\mathcal{D}] = \{(c_1, \ldots, c_n) \mid q(c_1, \ldots, c_n) \in M \text{ for each } M \in SM(\mathcal{P}[\mathcal{D}])\},$$

where n is the arity of Q. We simply refer to Q as $\mathcal{P}$ when q is clear from the context.

Example 2.2 In our ongoing example, we consider a query Q which asks for the codes of all players and team leaders, and which is formally written as $Q = \langle q, \mathcal{P} \rangle$ where $\mathcal{P} = \{q(x) \leftarrow player(x, y, z), q(x) \leftarrow team(v, w, x)\}$, and has arity 1. Note that $\mathcal{P}$ is a union of conjunctive queries. □

3 Data Integration and Query Answering Framework

In this section, we present an abstract framework for modeling query answering in data integration systems. 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 with respect to the integrity constraints specified on the global schema.

3.1 Data Integration Systems

According to [46], the formalization of 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. Notice that the above assumption implies that data stored at the sources are always considered locally consistent. This is a common assumption in data integration, because sources are in general external to the integration system, which is not in charge to analyze their consistency.

3. $\mathcal{M}$ is the *mapping* which establishes the relationship between $\mathcal{G}$ and $\mathcal{S}$. In our framework the mapping is given by the GAV approach, i.e., each global relation is associated with a *view*, i.e., a query, over the sources. We assume that the language used to express queries in the mapping is Datalog$^{\neg s}$, which allows us to generalize most of the GAV approaches proposed in the literature [46]. Therefore, $\mathcal{M}$ is a set of Datalog$^{\neg s}$ queries $Q = \langle r, \mathcal{P} \rangle$ over $\mathcal{S}$, one for each relation of $\mathcal{G}$, where r is a predicate of Ψ and $\mathcal{P}$ is a safe Datalog$^{\neg s}$ program with EDB predicates from Ψ' (cf. Example 1.1).

We call any database $\mathcal{DB}$ for the global schema $\mathcal{G}$ a *global database for* $\mathcal{I}$, and any database $\mathcal{D}$ for the source schema $\mathcal{S}$ a *source database for* $\mathcal{I}$. Then, based on $\mathcal{D}$, it is possible to compute a global database for $\mathcal{I}$ by exploiting the mapping specification.

Definition 3.1 Given a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ and a source database $\mathcal{D}$ for $\mathcal{I}$, the *retrieved global database*, $ret(\mathcal{I}, \mathcal{D})$, is the global database obtained by evaluating each query in the mapping $\mathcal{M}$ over $\mathcal{D}$, i.e., $ret(\mathcal{I}, \mathcal{D}) = \bigcup \{Q[\mathcal{D}] \text{ for each } Q \in \mathcal{M}\}$. □

In our running example, for instance, the retrieved global database is the global database of Figure 1.

Notice that $ret(\mathcal{I}, \mathcal{D})$ might violate 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 $\mathcal{I}$ [46]. Following a common approach in the literature on inconsistent databases [2, 36, 16], we then define the semantics of a data integration system $\mathcal{I}$ in terms of *repairs* of the database $ret(\mathcal{I}, \mathcal{D})$.

3.2 Database Repairs

We first focus on the setting of a single database, and consider the relational schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$, and a (possibly inconsistent) database $\mathcal{DB}$ for $\mathcal{RS}$. We suppose that $\leq_{\mathcal{DB}}$ is a preorder (i.e., a reflexive and transitive binary relation) on the set of all databases $\mathcal{DB}(\mathcal{RS})$, and denote by $<_{\mathcal{DB}}$ the naturally induced preference order (i.e., an irreflexive and transitive binary relation) given by $\mathcal{R}_1 <_{\mathcal{DB}} \mathcal{R}_2$, if $\mathcal{R}_1 \leq_{\mathcal{DB}} \mathcal{R}_2 \wedge \mathcal{R}_2 \not\leq_{\mathcal{DB}} \mathcal{R}_1$. We call $\mathcal{R}_1 \leq_{\mathcal{DB}}$-*preferred* to $\mathcal{R}_2$ in this case.

Then, we define the notion of repair of a database $\mathcal{DB}$ in terms of minimal elements of the preference order defined above.

Definition 3.2 (Repair) Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema, let $\mathcal{DB}$ be a database for $\mathcal{RS}$, and let $\leq_{\mathcal{DB}}$ be a preorder on subsets of $\mathcal{F}(\mathcal{RS})$. Then, a database $\mathcal{R} \in \mathcal{DB}(\mathcal{RS})$ is a *repair for* $\mathcal{DB}$ *w.r.t.* $\mathcal{RS}$, if

1. $\mathcal{R} \models \Sigma$, and
2. $\mathcal{R}$ is minimal w.r.t. $\leq_{\mathcal{DB}}$ among such sets, i.e., there exists no database $\mathcal{R}' \in \mathcal{DB}(\mathcal{RS})$ such that $\mathcal{R}' \models \Sigma$ and $\mathcal{R}'$ is $\leq_{\mathcal{DB}}$-preferred to $\mathcal{R}$.

The set of all repairs for $\mathcal{DB}$ w.r.t. $\mathcal{RS}$ is denoted by $rep_{\mathcal{RS}}(\mathcal{DB})$. □

When $\mathcal{RS}$ is clear from the context, the subscript $\mathcal{RS}$ may be dropped.

The definition of repair given above relies on a very general notion of a preorder on databases. The method for the evaluation of logic programs which we will present in the next sections is based on abstract properties of the induced preference order, which we refer to as set inclusion proximity and disjoint preference inheritance. The property of *set inclusion proximity* is as follows:

($\star$) For any databases $\mathcal{R}_1, \mathcal{R}_2$, and $\mathcal{DB}$, $\triangle(\mathcal{R}_1, \mathcal{DB}) \subset \triangle(\mathcal{R}_2, \mathcal{DB})$ implies $\mathcal{R}_1 <_{\mathcal{DB}} \mathcal{R}_2$,

where $\triangle(A, B) = (A \setminus B) \cup (B \setminus A)$ is symmetric set difference. Informally, it says that $\mathcal{R}$ can be minimal only if there is no way to establish consistency with Σ by touching merely a strict subset of facts compared to $\mathcal{R}$. The properties *disjoint preference expansion* and *disjunctive split* are follows:

(†) If $\mathcal{R}_1 <_{\mathcal{DB}_1} \mathcal{R}_1'$ and $\mathcal{R}_2, \mathcal{DB}_2$ are disjoint from $\mathcal{R}_1$, $\mathcal{R}_1'$, and $\mathcal{DB}_1$ (i.e., $(\mathcal{R}_1 \cup \mathcal{R}_1' \cup \mathcal{DB}_1) \cap (\mathcal{R}_2 \cup \mathcal{DB}_2) = \emptyset$), then $\mathcal{R}_1 \cup \mathcal{R}_2 <_{\mathcal{DB}_1 \cup \mathcal{DB}_2} \mathcal{R}_1' \cup \mathcal{R}_2$.

(‡) If $\mathcal{R}_1 <_{\mathcal{DB}} \mathcal{R}_2$, then for every database $\mathcal{R}$ it holds that either $\mathcal{R}_1 \cap \mathcal{R} <_{\mathcal{DB} \cap \mathcal{R}} \mathcal{R}_2 \cap \mathcal{R}$ or $\mathcal{R}_1 \setminus \mathcal{R} <_{\mathcal{DB} \setminus \mathcal{R}} \mathcal{R}_1' \setminus \mathcal{R}$.

Loosely speaking, Property (†) says that preference must be invariant under adding new facts, while Property (‡) says that preference must uniformly stem from disjoint "components."

Notice that a variety of repair semantics are either defined in terms of a preorder satisfying the above properties or can be characterized by such a preorder, including set-containment based ordering [32, 2, 4, 8, 14, 16, 19, 36, 34], cardinality-based [4, 53] ordering, weight-based ordering [52], as well as refinements with priority levels.

The prototypical preorder $\leq_{\mathcal{DB}}$ is given by $\mathcal{R}_1 \leq_{\mathcal{DB}} \mathcal{R}_2$ iff $\triangle(\mathcal{R}_1, \mathcal{DB}) \subseteq \triangle(\mathcal{R}_2, \mathcal{DB})$ [2, 4, 8, 14, 19, 36, 34]. Intuitively, each repair of $\mathcal{DB}$ is then obtained by properly adding and deleting facts from $\mathcal{DB}$ in order to satisfy constraints in Σ, as long as we "minimize" such additions and deletions. An interesting special case of weight-based ordering is the lexicographic preference, where $\mathcal{R}_1$ is preferred to $\mathcal{R}_2$ w.r.t. $\mathcal{DB}$ if the first fact in a total ordering of $\mathcal{F}(\mathcal{RS})$ on which $\mathcal{R}_1$ and $\mathcal{R}_2$ repair $\mathcal{DB}$ differently belongs to $\mathcal{R}_2$.

However, we point out that our method and results for query answering can also be extended to other preference orderings under certain conditions (see Section 8).

3.2.1 Safe constraints

An important aspect to address here is that constraints as introduced in Section 2.2 might enforce that *any* set of facts $\mathcal{R}$ for $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ which satisfies Σ must be infinite, and thus the database schema is inconsistent. A simple example is where Σ contains the constraint $\forall x\, p(x)$. Semantically, this is commonly avoided by requesting domain-independence of constraints [64], which syntactically is ensured by *safety* of the constraints Σ. Safety requests that each variable occurring in the head of a constraint must also occur in its body. If in addition to safe constraints the preference order $<_{\mathcal{DB}}$ satisfies set inclusion proximity, the existence of a repair $\mathcal{R}$ for $\mathcal{DB}$ w.r.t. $\mathcal{RS}$ is always guaranteed if $\mathcal{RS}$ is consistent. Moreover, $\mathcal{R}$ involves only constants which occur in $\mathcal{DB}$ or the constraints. The above property is formally stated by the following proposition, where, for any database $\mathcal{R} \subseteq \mathcal{F}(\mathcal{RS})$, we denote by $adom(\mathcal{R}, \mathcal{RS})$ the set of constants occurring in $\mathcal{R}$ and Σ.

Proposition 3.1 *Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema such that all constraints in Σ are safe, and let $\mathcal{DB}$ be a database for $\mathcal{RS}$. Suppose that $<_{\mathcal{DB}}$ satisfies Property ($\star$). Then, every repair $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$ involves only constants from $adom(\mathcal{DB}, \mathcal{RS})$, and some repair exists whenever $\mathcal{RS}$ is consistent.*

Proof. Let $\mathcal{R}$ be any database of $\mathcal{RS}$ consistent with Σ. Let $\mathcal{R}'$ result from $\mathcal{R}$ be removing every fact containing some constant $c \notin adom(\mathcal{DB}, \mathcal{RS})$. We show that $\mathcal{R}' \models \Sigma$. Towards a contradiction, assume that $\mathcal{R}' \not\models \Sigma$. Hence, there exists a ground instance σ^g of some constraint $\sigma \in \Sigma$ of form $A_1(\vec{c}_1) \wedge \cdots \wedge A_l(\vec{c}_l) \supset B_1(\vec{d}_1) \vee \cdots \vee B_m(\vec{d}_m) \vee \phi_1(\vec{e}_1) \vee \cdots \vee \phi_n(\vec{e}_n)$ which is violated by $\mathcal{R}'$, i.e., (i) $A_1(\vec{c}_1), \ldots, A_l(\vec{c}_l) \in \mathcal{R}'$, (ii) $B_1(\vec{d}_1), \ldots, B_m(\vec{d}_m) \notin \mathcal{R}'$, and (iii) $\phi_1(\vec{e}_1) \vee \cdots \vee \phi_n(\vec{e}_n)$ is false. Since $\mathcal{R} \models \sigma^g$, by construction of $\mathcal{R}'$ we have $B_j(\vec{d}_j) \in \mathcal{R} \setminus \mathcal{R}'$ for some $j \in \{1, \ldots, m\}$ and thus $\vec{d}_j$ contains some constant $c \notin adom(\mathcal{DB}, \mathcal{RS})$. It follows that some variable occurring in the head of σ does not occur in the body of σ; that is, σ is not safe, which is a contradiction.

Then, since $\mathcal{R}'$ is on $adom(\mathcal{DB}, \mathcal{RS})$, and $\triangle(\mathcal{R}', \mathcal{DB}) \subset \triangle(\mathcal{R}, \mathcal{DB})$ (since $R' \subset R$), $\leq_{\mathcal{DB}}$-minimality of repairs and Property ($\star$) imply that each repair $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$ must be on $adom(\mathcal{DB}, \mathcal{RS})$. Furthermore, by consistency of $\mathcal{RS}$ and the fact that each sequence $\mathcal{R}_1 >_{\mathcal{DB}} \mathcal{R}_2 >_{\mathcal{DB}} \cdots \mathcal{R}_i >_{\mathcal{DB}} \cdots$ of databases $\mathcal{R}_i$ on $adom(\mathcal{DB}, \mathcal{RS})$ must be finite, one such repair $\mathcal{R}$ must exist. □

Notice that major classes of constraints including key constraints, functional dependencies, exclusion dependencies, inclusion dependencies of the form $\forall \vec{x} p_1(\vec{x}) \supset p_2(\vec{x})$, or denial constraints fulfill safety.

We observe that finite repairs may be also ensured by unsafe constraints in which variables violating safety are guarded by built-in relations, such as for $\mathcal{DB} = \emptyset$ w.r.t. $\mathcal{RS} = \langle \{p\}, \{\forall x\, p(x) \vee x{>}100\} \rangle$, assuming that $\mathcal{U}$ are the natural numbers. As this example shows, repairs may in this case go beyond the active domain. This, however, is prevented if built-ins involve only equality and inequality. We have here a result similar to Proposition 3.1.

Proposition 3.2 *Let $\mathcal{RS} = \langle \Psi, \Sigma \rangle$ be a relational schema such that the constraints in Σ involve as built-in relations (if at all) only equality and inequality. Suppose that $<_{\mathcal{DB}}$ satisfies Property ($\star$). Then, every repair $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$ involves only constants from $adom(\mathcal{DB}, \mathcal{RS})$, and some repair exists whenever $\mathcal{RS}$ is consistent.*

Proof. The proof is similar to the one of Proposition 3.1. Following the argumentation there, some atom $B_j(\vec{c}_j) \in \mathcal{R} \setminus \mathcal{R}'$ in the head of a ground instance σ^g of some $\sigma \in \Sigma$ exists such that $\vec{c}_j = c_{j,1}, \ldots, c_{j,n_j}$ contains some constant $c_{j,h} \notin adom(\mathcal{DB}, \mathcal{RS})$ and the respective variable $y_{j,h}$ in the atom $B_j(\vec{y}_j)$ in σ does not occur in the body of σ. Since all built-in literals $\phi_1(\vec{z}_1), \ldots, \phi_n(\vec{z}_n)$ in σ are equalities and inequalities,

there are infinitely many constants c such that for the ground instance σ_c^g of σ which differs from σ^g only by substitution of $y_{j,h}$ with c, all built-in literals evaluate to false. Since σ_c^g and σ^g have the same body and $\mathcal{R} \models \sigma_c^g$ it follows that $\mathcal{R}$ must contain a fact in which c occurs. This means that $\mathcal{R}$ is infinite, which is a contradiction. □

The above propositions are important for a logic programming encoding of repairs, since they imply that domain closure can be applied when computing repairs. Using domain closure is customary with stable model semantics and engines like DLV and Smodels.

3.3 Repairs of a Data Integration System

Let us now turn to the data integration setting, and provide the notion of repair in this context.

Definition 3.3 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 $\mathcal{I}$. 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. $\mathcal{G}$. The set of all repairs for $\mathcal{I}$ w.r.t. $\mathcal{D}$ is denoted by $rep_{\mathcal{I}}(\mathcal{D})$. □

In the above definition we have implicitly considered the mapping $\mathcal{M}$ as *exact*, i.e., we have assumed that the data retrieved from the sources by 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 *completeness* assumptions [46]). 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., [17, 18, 16]).

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 (see Section 4.2 for an example).

3.4 Queries

A *query* over a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ is a query Q over $\mathcal{G}$. We assume that Q is a non-recursive Datalog$^\neg$ query. Note that in real integration applications, typically a language subsumed by non-recursive Datalog$^\neg$ is adopted. Given a database $\mathcal{D}$, the set of *consistent answers to Q over I w.r.t. $\mathcal{D}$* is the set of tuples

$$ans(Q, \mathcal{I}, \mathcal{D}) = \{t \mid t \in Q[\mathcal{R}] \text{ for each } \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D}) \}.$$

Informally, a tuple t is a consistent answer if it is a consequence under standard certainty semantics for each possible repair of the database $\mathcal{D}$.

Example 3.1 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 $team(RM, \textit{Roma}, 10)$ and $team(RM, \textit{Real Madrid}, 10)$. 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 $Q = \langle q, \mathcal{P} \rangle$, where $\mathcal{P} = \{q(x) \leftarrow player(x, y, z), q(x) \leftarrow team(v, w, x)\}$, we thus obtain that $ans(Q, \mathcal{I}_0, \mathcal{D}_0) = \{(8), (9), (10)\}$.

If we consider another query $Q' = \langle q', \mathcal{P}' \rangle$, with $P' = \{q'(y) \leftarrow team(x, y, z)\}$, we have that $ans(Q', \mathcal{I}_0, \mathcal{D}_0) = \{(\textit{Man. Utd.})\}$, while considering $Q'' = \langle q'', \mathcal{P}'' \rangle$, $\mathcal{P}'' = \{q''(x, z) \leftarrow team(x, y, z)\}$, we obtain that $ans(Q'', \mathcal{I}_0, \mathcal{D}_0) = \{(\textit{RM}, 10), (\textit{MU}, 8)\}$. □

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

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

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

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

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

7	*Camacho*	*RM*

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

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

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

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

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

7	*Camacho*	*RM*

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

4 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 raises.

4.1 Logic Programming Specification

According to several proposals in the literature [45, 12, 17, 14], we provide answers to user queries by encoding the mapping assertions in $\mathcal{M}$ and the constraints in Σ by means of 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 4.1 Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system where $\mathcal{G} = \langle \Psi, \Sigma \rangle$. Let $\mathcal{D}$ be a source database for $\mathcal{I}$, and let $Q = \langle q, \mathcal{P} \rangle$ be a non-recursive Datalog$^\neg$ query over $\mathcal{G}$. Then, a *logic specification for querying $\mathcal{I}$ with Q* is a safe Datalog$^{\vee,\neg}$ program $\Pi_{\mathcal{I}}(Q) = \Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_Q$ such that

1. $ret(\mathcal{I}, \mathcal{D}) \rightleftharpoons \mathrm{SM}(\Pi_{\mathcal{M}}[\mathcal{D}])$, where $\Pi_{\mathcal{M}}$ is a safe Datalog$^{\neg s}$ program,
2. $rep_{\mathcal{I}}(\mathcal{D}) \rightleftharpoons \mathrm{SM}(\Pi_{\Sigma}[ret(\mathcal{I}, \mathcal{D})])$, and
3. $ans(Q, \mathcal{I}, \mathcal{D}) = Q'[\mathcal{D}]$, where $Q' = \langle q, \Pi_{\mathcal{I}}(Q) \rangle$, i.e., $ans(Q, \mathcal{I}, \mathcal{D}) = \{t \mid q(t) \in M$ for each $M \in \mathrm{SM}((\Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_Q)[\mathcal{D}])\}$, where Π_Q is a non-recursive safe Datalog$^\neg$ program,

and $\rightleftharpoons$ denotes a polynomial-time computable correspondence between two sets. □

Notice that for the viability of such a logic specification in practice, it is assumed that repairs are on the active domain of the retrieved global database $ret(\mathcal{I}, \mathcal{D})$; this can be ensured by safety of constraints or limited use of built-ins (cf. discussion in Section 3.2).

The above definition establishes a connection between the semantics of $\Pi_{\mathcal{I}}(Q)$ and the consistent answers to a query Q posed to $\mathcal{I}$ (Item 3). In particular, $\Pi_{\mathcal{I}}(Q)$ is composed by three modules that can be hierarchically evaluated, i.e., $\Pi_{\mathcal{M}} \triangleright \Pi_{\Sigma} \triangleright \Pi_Q$ [26], using Splitting Sets [50]. More precisely,

- $\Pi_{\mathcal{M}}$ is used for retrieving data from the sources: the retrieved global database can be derived from its unique stable model (Item 1), provided some syntactic transformations, which typically are simple encodings such that $\rightleftharpoons$ is a linear-time computable bijection;
- Π_{Σ} is used for enforcing the constraints of Σ on the retrieved global database, whose repairs w.r.t. the global schema $\mathcal{G}$ can be derived from the stable models of $\Pi_{\Sigma}[ret(\mathcal{I}, \mathcal{D})]$ (Item 2). Correspondence between repairs and stable models is established again by a transformation $\rightleftharpoons$;
- finally, Π_Q is used for encoding the logic program $\mathcal{P}$ in the user query Q.

Our framework generalizes logic programming formalizations proposed in different integration settings, such as the ones presented in [45, 12, 17, 14]. 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 [36, 4, 8] 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 the mapping is assumed exact. The logic programs for querying a single database are of the form $\Pi_{\mathcal{I}}(Q) = \Pi_{\Sigma} \cup \Pi_Q$. Notice also that other logic-based approaches to data integration, based on abductive logic programming [5] and ID-logic [56], do not fit this framework.

4.2 Examples

We now consider some approaches for consistent query answering in inconsistent database and data integration systems that rely on the use of logic programming. We provide an example of logic program for each approach and show how it maps to the logic program specification for querying data integration systems given in Definition 4.1. The notion of repair adopted in the papers described below relies on the prototypical, natural preorder $\leq_{\mathcal{DB}}$, originally introduced in [2], for which $\mathcal{R}_1 \leq_{\mathcal{DB}} \mathcal{R}_2$ iff $\triangle(\mathcal{R}_1, \mathcal{DB}) \subseteq \triangle(\mathcal{R}_2, \mathcal{DB})$. The only exception is [17]. However, for the set of integrity constraints studied in [17] and considered in the following the adoption of a different repair semantics is of no concern. Logic programs that we devise in this section refer to the football team scenario introduced in Example 1.1.

4.2.1 Logic programs with unstratified negation

The paper [17] addresses the repair problem in GAV data integration systems in which key constraints are issued over the global schema, and presents a technique for consistent query answering based on the use of Datalog$^{\neg}$. According to [17], given a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, key constraints in $\mathcal{G}$ can be encoded into a suitable Datalog$^{\neg}$ program $\mathcal{P}_{KD}$, whereas views in the mapping, which are expressed as union of conjunctive queries, are casted into a Datalog program $\mathcal{P}_M$. Consistent answers to a union of conjunctive queries Q over $\mathcal{I}$ w.r.t. a source database $\mathcal{D}$ are returned by the evaluation of the Datalog$^{\neg}$ query $\langle q, \mathcal{P}_Q \cup \mathcal{P}_{KD} \cup \mathcal{P}_M \rangle$, where $\langle q, \mathcal{P}_Q \rangle$ is the Datalog encoding of the query Q.

In the following, we provide the logic program produced by the above technique for our running example. To this aim, we exploit an extension of the algorithm of [17], provided in [39], that allows for dealing with the exclusion dependencies specified in Example 1.1.

$$
\begin{aligned}
q(x) &\leftarrow player(x, y, z) \\
q(x) &\leftarrow team(v, w, x) \\
player_{\mathcal{D}}(x, y, z) &\leftarrow s_1(x, y, z, w) \\
team_{\mathcal{D}}(x, y, z) &\leftarrow s_2(x, y, z) \\
team_{\mathcal{D}}(x, y, z) &\leftarrow s_3(x, y, z) \\
coach_{\mathcal{D}}(x, y, z) &\leftarrow s_4(x, y, z) \\
player(x, y, z) &\leftarrow player_{\mathcal{D}}(x, y, z)\ ,\ not\ \overline{player}(x, y, z) \\
\overline{player}(x, y, z) &\leftarrow player(x, w, z)\ ,\ player_{\mathcal{D}}(x, y, z)\ ,\ y \neq v \\
team(x, y, z) &\leftarrow team_{\mathcal{D}}(x, y, z)\ ,\ not\ \overline{team}(x, y, z) \\
\overline{team}(x, y, z) &\leftarrow team(x, v, w)\ ,\ team_{\mathcal{D}}(x, y, z)\ ,\ y \neq v
\end{aligned}
$$

$$
\begin{aligned}
\overline{team}(x,y,z) &\leftarrow team(x,v,w)\ ,\ team_{\mathcal{D}}(x,y,z)\ ,\ z \neq w\\
coach(x,y,z) &\leftarrow coach_{\mathcal{D}}(x,y,z)\ ,\ not\ \overline{coach}(x,y,z)\\
\overline{coach}(x,y,z) &\leftarrow coach(x,w,z)\ ,\ coach_{D}(x,y,z)\ ,\ y \neq w\\
\overline{player}(x,y,z) &\leftarrow player_{\mathcal{D}}(x,y,z)\ ,\ coach(x,w,z)\\
\overline{coach}(x,y,z) &\leftarrow coach_{\mathcal{D}}(x,y,z)\ ,\ team(z,w,x)\\
\overline{coach}(x,y,z) &\leftarrow coach_{\mathcal{D}}(x,y,z)\ ,\ player(x,w,z)\\
\overline{team}(x,y,z) &\leftarrow team_{\mathcal{D}}(x,y,z)\ ,\ coach(z,w,x)
\end{aligned}
$$

In the above program, $\mathcal{P}_Q$ is constituted by the first two rules; $\mathcal{P}_M$ comprises the rules ranging from the 3rd to the 6th, $\mathcal{P}_{KD}$ the rules ranging from the 7th to the 15th, whereas the last four rules, which we denote by $\mathcal{P}_{ED}$, encode exclusion dependencies. Informally, for each global relation r, the above program contains (i) a relation $r_{\mathcal{D}}$ that represents $r^{ret(\mathcal{I},\mathcal{D})}$; (ii) a relation r that represents a subset of $r^{ret(\mathcal{I},\mathcal{D})}$ that is consistent with the key constraints and the exclusion dependencies for r; (iii) an auxiliary relation $\overline{r}$. It is easy to see that the above program constitutes the logic specification $\Pi_{\mathcal{I}_0}(Q) = \Pi_{\mathcal{M}_0} \cup \Pi_{\Sigma_0} \cup \Pi_Q$, where $\Pi_{\mathcal{M}_0} = \mathcal{P}_M$, $\Pi_\Sigma = \mathcal{P}_{KD} \cup \mathcal{P}_{ED}$, and $\Pi_Q = \mathcal{P}_Q$.

We point out that in [17], together with key constraints, also (existentially quantified) inclusion dependencies in the global schema $\mathcal{G}$ are considered. In this respect, a query reformulation technique is given that, on the basis of inclusion dependencies on $\mathcal{G}$, rewrites the user query Q into a new union of conjunctive queries Q_{ID}, again expressed over the global schema, in a way such that the consistent answers to Q over $\mathcal{I}$ w.r.t. a source database $\mathcal{D}$ coincide with consistent answers to Q_{ID} over $\mathcal{I}'$ w.r.t. $\mathcal{D}$, where $\mathcal{I}'$ is obtained from $\mathcal{I}$ by dropping the inclusion dependencies of $\mathcal{G}$. In other words, after computing Q_{ID}, it is possible to proceed as if inclusion dependencies were not specified on the global schema, i.e., by providing the logic specification for querying $\mathcal{I}'$ with Q_{ID} described above. Hence, after the first reformulation, the problem of computing consistent answers in the above setting and our problem coincide.

4.2.2 Logic programs with exceptions

A specification of database repairs for consistent query answering in inconsistent databases exploiting logic programs with exceptions (LPEs) is presented in [4]. We recall that this sort of programs, firstly introduced by [44], contains both *default rules*, i.e., classic clauses with classic negation in the body literals, and *exception rules*, i.e., clauses with negative heads whose conclusion overrides conclusions of default ones. Actually, [4] presents an extension of LPEs for accommodating both negative defaults and extended disjunctive exceptions whose semantics is given in terms of *e-answer sets*, and shows how these models are, in fact, in correspondence with standard stable models of a suitable standard disjunctive logic program.

In more detail, the transformation in [4] associates to each relation p in the database schema a new relation p' corresponding to its repaired version, and defines Π_Σ to contain three set of rules: *(i)* triggering exceptions, *(ii)* stabilizing exceptions, and *(iii)* persistence defaults. Let us, for instance, consider our running example. Then, triggering exception rules are as follows.

$$
\begin{aligned}
\neg\, player'(x,y,z) \vee \neg\, player'(x,y_1,z) &\leftarrow player(x,y,z),\ player(x,y_1,z),\ y \neq y_1.\\
\neg\, team'(x,y,z) \vee \neg\, team'(x,y_1,z_1) &\leftarrow team(x,y,z),\ team(x,y_1,z_1),\ y \neq y_1\\
\neg\, team'(x,y,z) \vee \neg\, team'(x,y_1,z_1) &\leftarrow team(x,y,z),\ team(x,y_1,z_1),\ z \neq z_1\\
\neg\, coach'(x,y,z) \vee \neg\, coach'(x,y_1,z) &\leftarrow coach(x,y,z),\ coach(x,y_1,z),\ y \neq y_1\\
\neg\, coach'(x,y,z) \vee \neg\, player'(x,y_1,z) &\leftarrow coach(x,y,z),\ player(x,y_1,z)
\end{aligned}
$$

$$\neg\, coach'(x,y,z) \vee \neg\, team'(z,y_1,x) \;\leftarrow\; coach(x,y,z),\; team(z,y_1,x)$$

The above rules represent a suitable rewriting of the integrity constraints that encodes the basic way of repairing each inconsistency. For example, a conflict on a key is resolved by deleting one of the tuples that cause the conflict, i.e., by not including this tuple in the extension of the corresponding primed predicate. Notice that, in the case of (universally quantified) inclusion dependencies, it is possible to have repairs by adding tuples. For instance, the constraint $p(x,y) \supset q(x,y)$ would be repaired with the rule

$$\neg\, p'(x,y) \vee q'(x,y) \leftarrow p(x,y), not\; q(x,y).$$

Stabilizing exception rules and persistence defaults have been introduced for technical reasons. Indeed, rules of the former kind state that each integrity constraint must be eventually satisfied in the repair while rules of the latter kind impose that by default each relation p' contains the facts in p.

Given the rewriting Π_Σ, the user query can be simply issued over the primed relations, i.e., the program Π_Q is easily obtained by substituting in the user query (suitably expressed in Datalog) each predicate p with its repaired version p'. Notice that the framework proposed in [4] can be easily adapted to deal with GAV data integration systems with exact mappings: basically, it is sufficient to add the Datalog specification of the GAV mapping to $\Pi_\Sigma \cup \Pi_Q$.

4.2.3 Annotated Logic

The paper [8] proposes to specify database repairs by means of disjunctive normal programs under the stable model semantics. To this aim, suitable annotations are used in an extra argument introduced in each (non built-in) predicate of the logic program, for marking the operations of insertion and deletion of tuples required in the repair process. The idea of annotating predicates has been inspired by the Annotated Predicate Calculus [43], a non-classical logic in which inconsistencies may be accommodated without trivializing reasoning. The values used in [8] for the annotations are:

- $\mathbf{t_d}$ and $\mathbf{f_d}$, which indicate whether, before the repair, a given tuple is in the database or not, respectively;
- $\mathbf{t_a}$ and $\mathbf{f_a}$, which represent advisory values that indicate how to resolve possible conflicts, i.e., a tuple annotated with $\mathbf{t_a}$ (resp. $\mathbf{f_a}$) has to be inserted (resp. deleted) in the database;
- $\mathbf{t}^*$ and $\mathbf{f}^*$, which indicate whether a given tuple is in the repaired database or not, respectively.

For instance, the annotated logic program used for solving the conflicts on the key of the relation $player$ in our running example is as follows:

$$\begin{aligned}
player(x,y,z,\mathbf{t}^*) &\leftarrow player(x,y,z,\mathbf{t_d}) \\
player(x,y,z,\mathbf{t}^*) &\leftarrow player(x,y,z,\mathbf{t_a}) \\
player(x,y,z,\mathbf{f}^*) &\leftarrow not\; player(x,y,z,\mathbf{t_d}) \\
player(x,y,z,\mathbf{f}^*) &\leftarrow player(x,y,z,\mathbf{f_a}) \\
player(x,y,z,\mathbf{f_a}) \vee player(x,y_1,z,\mathbf{f_a}) &\leftarrow player(x,y,z,\mathbf{t}^*),\; player(x,y_1,z,\mathbf{t}^*),\; y \neq y_1.
\end{aligned}$$

Furthermore, each fact in the original database is assumed to be annotated by $\mathbf{t_d}$.

Intuitively, the last rule says that when the key of the relation $player$ is violated (body of the rule), the database instance has to be repaired according to one of the two alternatives shown in the head. Possible

interaction between different constraints are then taken into account by the other rules, which force the repair process to continue and stabilize in a state in which all the integrity constraints hold. Indeed, annotations $\mathbf{t}^*$ and $\mathbf{f}^*$ can feed back rules of the last kind, until consistency is restored. This should be evident if we consider also a constraint of the form $coach(x, y, z) \supset player(x, y, z)$ (we disregard exclusion dependencies of our running example for a while). This constraint is repaired with the rule

$$coach(x, y, z, \mathbf{f_a}) \vee player(x, y, z, \mathbf{t_a}) \leftarrow coach(x, y, z, \mathbf{t}^*), player(x, y, z, \mathbf{f}^*),$$

besides the rules for the predicate $coach$ that compute facts with annotations $\mathbf{t}^*$ (resp. $\mathbf{f}^*$) from facts annotated by $\mathbf{t_d}$ or $\mathbf{t_a}$ (resp. $\mathbf{f_d}$ or $\mathbf{f_a}$).

The program Π_Q is then computed by reformulating the original query according to the annotations: in our running example, we have

$$q(x) \leftarrow player(x, y, z, \mathbf{t_a}) \vee (player(x, y, z, \mathbf{t_d}) \wedge \neg player(x, y, z, \mathbf{f_a}))$$
$$q(x) \leftarrow team(v, w, x, \mathbf{t_a}) \vee (team(v, w, x, \mathbf{t_d}) \wedge \neg team(v, w, x, \mathbf{f_a})).$$

The above rewriting is proposed in [8] for the setting of a single database but can be straightforwardly extended to work in GAV data integration systems. An interesting, more complex generalization to the LAV setting appears instead in [14, 15]. Since in LAV each source relation is associated with a query over the global schema, an exact specification of which data of the sources fit the global schema is actually missing. In general, given a source database, several different ways of populating the global schema according to the mapping may exist. Hence, not a single but multiple retrieved global databases have be taken into account for repairing. According to [14, 15], the repairs are defined as those consistent global databases which have a minimal (under set inclusion) symmetric difference to one of the minimal (again, under set inclusion) retrieved global databases. In other words, each such retrieved global database is repaired by adopting the classic preorder of [2]. These repairs can be obtained from the stable models of a suitable disjunctive logic program, which comprises rules for the encoding of integrity constraints constructed as in [8], and specific rules for computing the minimal retrieved global databases.

5 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. Indeed, whereas both $\Pi_\mathcal{M}$, which is in general a Datalog$^{\neg s}$ program, and Π_Q, which is in general a non-recursive Datalog$^\neg$ program, can be evaluated in polynomial time with respect to underlying databases (data complexity) [22], Π_Σ is in general a Datalog$^{\vee,\neg}$ program [36], whose evaluation data complexity is at the second level of the polynomial hierarchy [22]. Furthermore, also evaluating programs with lower complexity over large data sets by means of stable models solvers, quickly becomes infeasible. This calls for suitable optimization methods speeding up the evaluation (as recently stated in [14]).

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, which has to be performed over each repair of the retrieved global database.

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 in Section 5.2) by any violation. Then, we obtain the repairs of $ret(\mathcal{I}, \mathcal{D})$ by computing the repairs of the affected facts, and by then adding to each such repair the unaffected facts. Finally, we suitably recombine the repairs of $ret(\mathcal{I}, \mathcal{D})$ to provide consistent answers to user queries.

Notice that computing the repairs of the set of affected facts means in practice evaluating the program Π_Σ only over this set, rather than on the whole retrieved global database, as needed when directly computing the repairs of $ret(\mathcal{I}, \mathcal{D})$ in the evaluation of the program $\Pi_\mathcal{I}(Q)$ over the source database $\mathcal{D}$ (Item 2 in Definition 4.1). Since, in general, the size of the set of the affected facts is much smaller than the size of the retrieved global database, this way of proceeding is significantly faster than the naive evaluation of $\Pi_\mathcal{I}(Q)$. For this to work, we assume that the preference orders $<_{\mathcal{DB}}$ satisfy the properties ($\star$), (†), and (‡) from Section 3.

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

Relevance 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 $\Pi_\mathcal{I}(Q)$, and it is not a crucial aspect of our technique.

Decomposition: We localize inconsistency in the retrieved global database, and single out facts that are affected by repair, and facts that are not. Finally, we compute repairs of the set of affected facts, and from them we obtain repairs of the retrieved global database, by incorporating non affected facts.

Recombination: We suitably recombine the repairs of the retrieved global database for computing the answers to Q.

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

5.1 Relevance Pruning

In this step, we localize the portion of $ret(\mathcal{I}, \mathcal{D})$ that is needed to answer the query Q, thus avoiding to retrieve from the sources tuples that do not contribute to answering the query; moreover, we also select the rules of $\Pi_\mathcal{I}(Q)$ that are necessary for handling constraint violations in such a portion of the retrieved global database, thus disregarding non relevant rules.

It is worthwhile noting that the relevance pruning phase relies on principles that are quite close to those exploited by the magic-set technique [6, 9]. Indeed, this phase is aimed at singling out a (small) portion of the logic specification which suffices for evaluating a given user query, by identifying predicates and rules which are relevant for answering user queries. Actually, in the magic-set technique the notion of relevance has been "syntactically" defined on the basis of some structural properties of the programs at hand, in order to have a kind of general optimization strategy. Conversely, in our approach the notion of relevance will be "semantically" defined on the basis of the integrity constraints issued over the global schema, thereby taking advantage of the specific application domain.

Relevance pruning is carried out by preliminarily singling out the relation predicates that are relevant for answering a given user query.

Definition 5.1 Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$, and Q a query over $\mathcal{I}$. Then, the set of *relevant relation symbols w.r.t.* Q is the smallest set of relation symbols r in Ψ such that

(a) r is a predicate in the body of the rules of Q, or

(b) r is involved in a dependency in Σ with a relation s relevant w.r.t. Q. □

We denote by $ret_r(\mathcal{I}, \mathcal{D}, Q)$ the subset of $ret(\mathcal{I}, \mathcal{D})$ that contains only the extension of relations that are relevant w.r.t. Q, i.e., $ret_r(\mathcal{I}, \mathcal{D}, Q) = \{r(t) \in ret(\mathcal{I}, \mathcal{D}) \mid r \text{ is relevant w.r.t. } Q\}$.

Example 5.1 Consider again the query $Q = \langle q, \mathcal{P} \rangle$ in Example 2.2, where

$$\mathcal{P} = \{q(x) \leftarrow player(x, y, z), q(x) \leftarrow team(v, w, x)\}.$$

Then, the relations $player$ and $team$ are relevant due to condition (a), whereas relation $coach$ is relevant due to condition (b), witnessed by the constraints (dropping quantifiers) $coach(x, y, z) \wedge player(x', y', z) \supset x \neq x'$ and $coach(x, y, z) \wedge team(z, y', x') \supset x \neq x'$. Hence, in this case $ret_r(\mathcal{I}_0, \mathcal{D}_0, Q) = ret(\mathcal{I}_0, \mathcal{D}_0)$. □

Armed with the above notions, we next define which is the portion of $\Pi_{\mathcal{I}}(Q)$ that can be considered relevant for computing consistent answers to Q w.r.t. $\mathcal{D}$.

Definition 5.2 Given a data integration system $\mathcal{I}$, and a logic program specification $\Pi_{\mathcal{I}}(Q)$ for $\mathcal{I}$, we say that a rule ρ in $\Pi_{\mathcal{I}}(Q)$ is *relevant w.r.t.* Q, if ρ contains a literal expressed in terms of a relation symbol r that is relevant w.r.t. Q, or predicates occurring in ρ occur also in a rule ρ' that is relevant w.r.t. Q. We denote by $\Pi_{\mathcal{I}r}(Q)$ the set of rules in $\Pi_{\mathcal{I}}(Q)$ that are relevant w.r.t. Q. □

Obviously, the above definitions are well-founded if evaluating $\Pi_{\mathcal{I}r}(Q)$ over $ret_r(\mathcal{I}, \mathcal{D}, Q)$ is sufficient to compute consistent answers to Q over $\mathcal{I}$ w.r.t. $\mathcal{D}$. However, the soundness of the relevance pruning phase strongly depends on the particular structure of the logic specification $\Pi_{\mathcal{I}}(Q) = \Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_Q$ used for querying $\mathcal{I}$. And, in fact, posed $Q = \langle q, \mathcal{P} \rangle$, for most of the logic specification in the literature, it is not difficult to prove that

$$ans(Q, \mathcal{I}, \mathcal{D}) \rightleftharpoons \{t \mid q(t) \in M \text{ for each } M \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})_r[ret_r(\mathcal{I}, \mathcal{D}, Q)])\}$$

where $(\Pi_Q \cup \Pi_{\Sigma})_r$ indicates the relevant portion of $\Pi_Q \cup \Pi_{\Sigma}$. To this aim, we first recall that, by Definition 4.1,

$$ans(Q, \mathcal{I}, \mathcal{D}) \rightleftharpoons \{t \mid q(t) \in M \text{ for each } M \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})[ret(\mathcal{I}, \mathcal{D})])\}.$$

Hence, in order to establish soundness of relevance pruning, it is sufficient to show that the following two properties hold:

R1 $\forall M \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})_r[ret_r(\mathcal{I}, \mathcal{D}, Q)]), \exists M' \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})[ret(\mathcal{I}, \mathcal{D})])$ such that $M|q = M'|q$

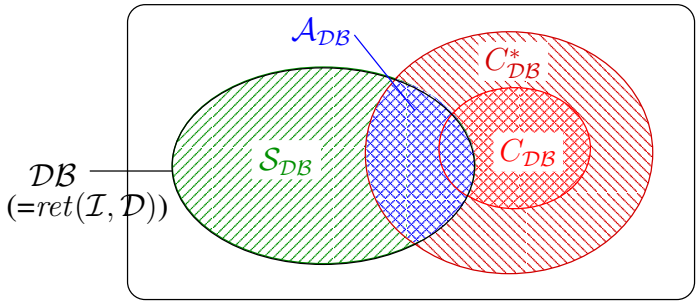
R2 $\forall M' \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})[ret(\mathcal{I}, \mathcal{D})]), \exists M \in \mathrm{SM}((\Pi_Q \cup \Pi_{\Sigma})_r[ret_r(\mathcal{I}, \mathcal{D}, Q)])$ such that $M|q = M'|q$

where $S|p = \{p(t) \in S\}$ for any set of facts S and predicate p.

Consider the programs $\mathcal{P} = (\Pi_Q \cup \Pi_{\Sigma})[ret(\mathcal{I}, \mathcal{D})]$ and $\mathcal{P}_r = (\Pi_Q \cup \Pi_{\Sigma})_r[ret_r(\mathcal{I}, \mathcal{D}, Q)])$, and notice that $\mathcal{P}_r \subseteq \mathcal{P}$. Then, let $\mathcal{P}_{nr}$ result from $\mathcal{P}$ by deleting all rules in $\mathcal{P}_r$. For most of the logic specifications in the literature, e.g., for those considered in the previous section, it is easy to see that by construction, (i) $\mathcal{P}_{nr}$ is a Splitting Set [50] for $\mathcal{P}$ and (ii) $\mathrm{SM}(\mathcal{P}) \neq \emptyset$. Then, both parts $\mathcal{P}_r$ and $\mathcal{P}_{nr}$ have stable models and in fact $\mathrm{SM}(\mathcal{P}) = \bigcup\{\mathrm{SM}(\mathcal{P}_{nr} \cup M) \mid M \in \mathrm{SM}(\mathcal{P}_r)\}$. Then, properties R1 and R2 are straightforward from the fact that the predicate q is defined in the module $\mathcal{P}_r$.

Armed with this result, we implement the relevance pruning step by computing the program $\Pi_{\mathcal{I}r}(Q)$ and $ret_r(\mathcal{I}, \mathcal{D}, Q)$. Notice that the cost of this step is dominated by the effort required for computing the relation symbols that are relevant w.r.t Q and does not require any database scan. Indeed, to this aim we need to apply Definition 5.1, whose cost is quadratic in the size of the schema.

In the following, we implicitly consider $ret_r(\mathcal{I}, \mathcal{D}, Q)$ and $\Pi_{\mathcal{I}r}(Q)$. Nonetheless, to keep things simple, we do not use the subscript r, and simply refer to $ret(\mathcal{I}, \mathcal{D})$ and $\Pi_{\mathcal{I}}(Q)$.



Conflict set $\mathcal{C}_{\mathcal{DB}}$: facts occurring in $ground(\Sigma)$ violated in $\mathcal{DB}$.
Conflict closure $\mathcal{C}^*_{\mathcal{DB}}$: syntactic conflict propagation by Σ

$\mathcal{S}_{\mathcal{DB}}$ = $\mathcal{DB} \setminus \mathcal{C}_{\mathcal{DB}}{}^*$ is the *safe database* for $\mathcal{DB}$.
$\mathcal{A}^*_{\mathcal{DB}}$ = $\mathcal{DB} \cap \mathcal{C}_{\mathcal{DB}}{}^*$ is the *affected database* for $\mathcal{DB}$.

Figure 3: Decomposition for database repair

We conclude this section by pointing out that the solution we have provided can be extended and optimized in several directions. Firstly, the notion of relevance may be refined such that arbitrary logic specifications of repairs can be handled, assuming that repairs exist, cf. [24]. Secondly, the pruning step strongly depends on the user query and the form of constraints on the global schema, and proper solutions for particular and significant classes of queries and constraints might be adopted. For instance, the case in which only key constraints are specified on the relations involved in the query can be basically faced by computing only the extension of the EDB predicates occurring in the query Q. However, since pruning techniques are not the main focus of the paper, we have introduced a general technique which can be eventually improved by exploiting some additional knowledge on the type of constraints issued over the global schema.

5.2 Decomposition

In this section, we investigate how to localize inconsistency in the retrieved global database, i.e., compute the set of facts that will be possibly touched by repair, called affected database, and how to obtain all repairs of the retrieved global database from the repairs of the affected database. We start with some concepts for a single database, which are illustrated in Figure 3.

Let $\mathcal{DB}$ be a database for a relational schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$. Then, the *conflict set* for $\mathcal{DB}$ w.r.t. $\mathcal{RS}$ is the set of facts $\mathcal{C}^{\mathcal{RS}}_{\mathcal{DB}} = \{p(t) \mid \exists \sigma^g \in ground(\Sigma), p(t) \in facts(\sigma^g), \mathcal{DB} \not\models \sigma^g\}$, i.e., $\mathcal{C}^{\mathcal{RS}}_{\mathcal{DB}}$ is the set of facts occurring in the ground instances of Σ which are violated by $\mathcal{DB}$. In the following, if clear from the context, we shall drop the superscript $\mathcal{RS}$.

Example 5.2 In our ongoing example, let $\mathcal{DB} = ret(\mathcal{I}_0, \mathcal{D}_0)$. Then, the conflict set consists of the facts that violate the key constraints on $team$, i.e., $\mathcal{C}_{\mathcal{DB}} = \{team(RM, \textit{Roma}, 10), team(RM, \textit{Real Madrid}, 10)\}$. □

Figure 3 shows that the conflict set may contain both facts of the database $\mathcal{DB}$ (as in Example 5.2) and facts of $\mathcal{F}(\mathcal{RS})$ that do not belong to $\mathcal{DB}$. For example, let $\mathcal{DB} = \{p(a)\}$, and let $\mathcal{RS}$ contain the dependency $\forall x p(x) \supset q(x)$. Then, we have that $\mathcal{C}_{\mathcal{DB}} = \{p(a), q(a)\}$.

Notice that, in general, the notion of conflict set is not sufficient to localize inconsistency in a database instance. Actually, we must take also care of those facts of $\mathcal{F}(\mathcal{RS})$ that "indirectly" take part to constraint

violations. To this aim, we introduce the notion of constraint-bounded facts: two facts $p(t), p'(t')$ in $\mathcal{F}(\mathcal{RS})$ are *constraint-bounded in* $\mathcal{RS}$, if there exists some $\sigma^g \in ground(\Sigma)$ such that all constants occurring in $facts(\sigma^g)$ are from $adom(\mathcal{DB}, \mathcal{RS})$, and $\{p(t),\ p'(t')\} \subseteq facts(\sigma^g)$. (Note that by assumed safety of constraints and the results of Section 3.2.1, we only need to consider $adom(\mathcal{DB}, \mathcal{RS})$.) Then, we provide the following definition.

Definition 5.3 (Conflict closure; safe and affected database) 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 $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$, is the least set $\mathcal{C} \supseteq \mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}}$ which contains every fact $p(t)$ constraint-bounded in $\mathcal{RS}$ with some fact $p'(t') \in \mathcal{C}$. Moreover, we call $\mathcal{S}_{\mathcal{DB}}^{\mathcal{RS}^*} = \mathcal{DB} \setminus \mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$ and $\mathcal{A}_{\mathcal{DB}}^{\mathcal{RS}^*} = \mathcal{DB} \cap \mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$ the *safe database* and the *affected database for* $\mathcal{DB}$, respectively. □

As shown in Figure 5.2, $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$ may add to $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}}$ both facts in $\mathcal{DB}$ or outside $\mathcal{DB}$. Assume for instance that $\mathcal{RS}$ in the example above contains also the constraint $\forall x q(x) \supset s(x)$. Then, we have that $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*} = \mathcal{C}_{\mathcal{DB}} \cup \{s(a)\}$. Notice, however, that $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}}$ and $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$ would have been the same even if $s(a)$ was in $\mathcal{DB}$. In the following, we drop the superscript $\mathcal{RS}$ from $\mathcal{C}_{\mathcal{DB}}^{\mathcal{RS}^*}$, $\mathcal{S}_{\mathcal{DB}}^{\mathcal{RS}^*}$ and $\mathcal{A}_{\mathcal{DB}}^{\mathcal{RS}^*}$ if $\mathcal{RS}$ is clear from the context.

Intuitively, $\mathcal{C}_{\mathcal{DB}}^*$ contains facts belonging to $\mathcal{C}_{\mathcal{DB}}$, which therefore cause inconsistency, and facts which possibly must be touched by repair in turn to avoid new inconsistency with Σ caused by previous repairing actions. $\mathcal{A}_{\mathcal{DB}}^*$ is the portion of $\mathcal{C}_{\mathcal{DB}}^*$ contained in $\mathcal{DB}$, whereas $\mathcal{S}_{\mathcal{DB}}^*$ is the portion of $\mathcal{DB}$ that contains facts not involved in constraint violations and that for sure will not be touched by repair.

Example 5.3 The facts $team(RM, \textit{Roma}, 10)$ and $team(RM, \textit{Real Madrid}, 10)$ are trivially in $\mathcal{C}_{\mathcal{DB}}^*$, since they are in $\mathcal{C}_{\mathcal{DB}}$. Moreover, among other facts, $\mathcal{C}_{\mathcal{DB}}^*$ contains, for example, $coach(7, \textit{Camacho}, RM)$ and $player(10, \textit{Totti}, RM)$, where the former is constraint-bounded, e.g., with $team(RM, \textit{Roma}, 10)$ due to the ground constraint $coach(7, \textit{Camacho}, RM) \wedge team(team(RM, \textit{Roma}, 10) \supset 7{\neq}10$, and the latter is constraint-bounded with $coach(7, \textit{Camacho}, RM)$ due to the ground constraint $coach(7, \textit{Camacho}, RM) \wedge player(10, \textit{Totti}, RM) \supset 7{\neq}10$. Then, it is easy to see that, in this case,

$$\begin{aligned}\mathcal{A}_{\mathcal{DB}}^* &= \{team(RM, \textit{Roma}, 10), team(RM, \textit{Real Madrid}, 10),\\ &\quad\ coach(7, \textit{Camacho}, RM), player(10, \textit{Totti}, RM)\},\ \text{and}\\ \mathcal{S}_{\mathcal{DB}}^* &= \{player(9, \textit{Beckham}, MU)\}.\end{aligned}$$

Notice that, in this example, $\mathcal{A}_{\mathcal{DB}}{}^*$ contains facts as $coach(7, \textit{Camacho}, RM)$ and $player(10, \textit{Totti}, RM)\}$ that actually do not occur in any ground constraint which is violated by the facts in $\mathcal{C}_{\mathcal{DB}}^*$. Even if this might seem counterintuitive, at the end of this section we will show that, for the kind of integrity constraints specified on the database schema, in this case (and in many other practical cases) there is no need to take into account such facts in order to repair the database, and that we can focus simply on $\mathcal{C}_{\mathcal{DB}}$. However, in general it is necessary to resort to the computation of $\mathcal{C}_{\mathcal{DB}}^*$ which may contain also facts apparently not "dangerous", but which indirectly participate in constraint violations since are (iteratively) constraint-bounded with some facts which cause a conflict. □

Since in practice the size of $\mathcal{A}_{\mathcal{DB}}^*$ is expected to be much smaller than the size of $\mathcal{DB}$, the idea of our approach is to focus the computation of the repairs to $\mathcal{A}_{\mathcal{DB}}^*$ only, and then computing the repairs of $\mathcal{DB}$ by adding the safe database $\mathcal{S}_{\mathcal{DB}}^*$ to each such repair. As we will see, this can be efficiently done in many practical cases.

Before presenting the results that formally allow us to pursue our strategy, we need some preliminary technical results. Consider the following two subsets of all ground constraints:

(i) $\Sigma^a_{\mathcal{DB}} = \{\sigma^g \in ground(\Sigma) \mid facts(\sigma^g) \cap \mathcal{C}^*_{\mathcal{DB}} \neq \emptyset\}$ consists of all the ground constraints in which at least one fact from $\mathcal{C}^*_{\mathcal{DB}}$ occurs;

(ii) $\Sigma^s_{\mathcal{DB}} = \{\sigma^g \in ground(\Sigma) \mid facts(\sigma^g) \not\subseteq \mathcal{C}^*_{\mathcal{DB}}\}$ consists of all the ground constraints in which at least one fact occurs which is *not* in $\mathcal{C}^*_{\mathcal{DB}}$.

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

Proposition 5.1 (Separation) *Let $\mathcal{DB}$ be a database for a relational schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$. Then,*

1. $facts(\Sigma^a_{\mathcal{DB}}) \subseteq \mathcal{C}^*_{\mathcal{DB}}$;
2. $facts(\Sigma^s_{\mathcal{DB}}) \cap \mathcal{C}^*_{\mathcal{DB}} = \emptyset$;
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. By definition of $\Sigma^a_{\mathcal{DB}}$, $\sigma^g \in \Sigma^a_{\mathcal{DB}}$ contains at least one fact $p(t)$ from $\mathcal{C}^*_{\mathcal{DB}}$; any other fact in σ^g is constraint-bounded in $\mathcal{RS}$ with $p(t)$, and hence it also must be in $\mathcal{C}^*_{\mathcal{DB}}$.

2. Assume by contradiction that some $\sigma^g \in \Sigma^s_{\mathcal{DB}}$ with $facts(\sigma^g) \cap \mathcal{C}^*_{\mathcal{DB}} \neq \emptyset$ exists. Then, Definition 5.3 implies $facts(\sigma^g) \subseteq \mathcal{C}^*_{\mathcal{DB}}$, which contradicts $\sigma^g \in \Sigma^s_{\mathcal{DB}}$.

3. Part 3 is straightforward from Part 1 and Part 2. □

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

Proposition 5.2 (Safe database) *Let $\mathcal{DB}$ be a database for a relational schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$. Then, for each repair $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$, it holds that $\mathcal{S}^*_{\mathcal{DB}} = \mathcal{R} \setminus \mathcal{C}^*_{\mathcal{DB}}$.*

Proof. Towards a contradiction, suppose that there exists a repair $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$ such that $\mathcal{R} \setminus \mathcal{C}^*_{\mathcal{DB}} \neq \mathcal{S}^*_{\mathcal{DB}}$. Let $\mathcal{R}' = (\mathcal{R} \cap \mathcal{C}^*_{\mathcal{DB}}) \cup (\mathcal{DB} \setminus \mathcal{C}^*_{\mathcal{DB}})$. Consider any $\sigma^g \in ground(\Sigma)$. By Proposition 5.1, either (i) $\sigma^g \in \Sigma^a_{\mathcal{DB}}$ or (ii) $\sigma^g \in \Sigma^s_{\mathcal{DB}}$. Moreover, since $\mathcal{R} \models \sigma^g$, it follows that in case (i) $\mathcal{R}' \models \sigma^g$. Similarly, since $\mathcal{S}^*_{\mathcal{DB}} \models \sigma^g$ in case (ii) $\mathcal{R}' \models \sigma^g$. Thus, $\mathcal{R}' \models \Sigma$. However, $\mathcal{R}'$ differs from $\mathcal{R}$ only in the tuples which are are not in $\mathcal{C}^*_{\mathcal{DB}}$, and thus $\triangle(\mathcal{R}', \mathcal{DB}) \subset \triangle(\mathcal{R}, \mathcal{DB})$. From Property $(\star)$, it follows $\mathcal{R}' <_{\mathcal{DB}} \mathcal{R}$. This contradicts $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{DB})$. □

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

Lemma 5.3 *Let $\mathcal{DB}$ be a database for a relational schema $\mathcal{RS} = \langle \Psi, \Sigma \rangle$, and let $\mathcal{RS}^a_{\mathcal{DB}} = \langle \Psi, \Sigma^a_{\mathcal{DB}} \rangle$. Then, for each $\mathcal{S} \subseteq \mathcal{S}^*_{\mathcal{DB}}$, the following holds:*

1. *for each $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$, $(\mathcal{R} \cap \mathcal{C}^*_{\mathcal{DB}}) \in rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$;*
2. *for each $\mathcal{R}_a \in rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$ there exists a set of facts $\mathcal{S}' \subseteq \mathcal{F}(\mathcal{RS}) \setminus \mathcal{C}^*_{\mathcal{DB}}$, such that $(\mathcal{R}_a \cup \mathcal{S}') \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$.*

Proof.

1. Let $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$, and let $\mathcal{R}_a = \mathcal{R} \cap \mathcal{C}^*_{\mathcal{DB}}$. Since $\mathcal{R} \models \Sigma$, Proposition 5.1 implies that $\mathcal{R}_a \models \Sigma^a_{\mathcal{DB}}$, while $\mathcal{R} \setminus \mathcal{R}_a \models \Sigma^s_{\mathcal{DB}}$. Assume $\mathcal{R}_a \notin rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$. Then, there exists some $\mathcal{R}'_a \in rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$ such that $\mathcal{R}'_a <_{\mathcal{A}^*_{\mathcal{DB}}} \mathcal{R}_a$. Since $\mathcal{R}'_a \models \Sigma^a_{\mathcal{DB}}$ and $\mathcal{R} \setminus \mathcal{R}_a \models \Sigma^s_{\mathcal{DB}}$, we have that $\mathcal{R}'_a \cup (\mathcal{R} \setminus \mathcal{R}_a) \models \Sigma$. Since $\mathcal{C}^{\mathcal{RS}^*}_{\mathcal{DB}} = \mathcal{C}^{\mathcal{RS}^a_{\mathcal{DB}}{}^*}_{\mathcal{DB}}$ and $\mathcal{S}^{\mathcal{RS}^a_{\mathcal{DB}}{}^*}_{\mathcal{A}_{\mathcal{DB}}} = \emptyset$, by Proposition 5.2 we have $\mathcal{R}'_a \subseteq \mathcal{C}^{\mathcal{RS}^*}_{\mathcal{DB}}$. By Property (†), it then follows that $\mathcal{R}'_a \cup (\mathcal{R} \setminus \mathcal{R}_a) <_{\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S}} \mathcal{R}_a \cup (\mathcal{R} \setminus \mathcal{R}_a) = \mathcal{R}$. This contradicts that $\mathcal{R} \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$.

2. Let $\mathcal{R}_a \in rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$. We show that there exists some $\mathcal{R}_{as} \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$ of form $\mathcal{R}_{as} = \mathcal{R}_a \cup \mathcal{S}'$ such that $\mathcal{R}_{as} \cap \mathcal{A}^*_{\mathcal{DB}} = \mathcal{R}_a$. Indeed, consider an arbitrary repair $\mathcal{S}' \in rep_{\mathcal{RS}^s_{\mathcal{DB}}}(\mathcal{S})$ where $\mathcal{RS}^s_{\mathcal{DB}} = \langle \Psi, \Sigma^s_{\mathcal{DB}} \rangle$. By Proposition 5.2, the safe part $\mathcal{S}^{\mathcal{RS}^s_{\mathcal{DB}}{}^*}_{\mathcal{S}}$ of $\mathcal{S}$ w.r.t. $\mathcal{RS}^s_{\mathcal{DB}}$ satisfies $\mathcal{S}^{\mathcal{RS}^s_{\mathcal{DB}}{}^*}_{\mathcal{S}} = \mathcal{S}' \setminus \mathcal{C}^{\mathcal{RS}^s_{\mathcal{DB}}{}^*}_{\mathcal{S}}$. Since $\mathcal{S}^{\mathcal{RS}^s_{\mathcal{DB}}{}^*}_{\mathcal{S}} \subseteq \mathcal{S}$ and clearly $C^{\mathcal{RS}^*}_{\mathcal{DB}}$ is disjoint from $\mathcal{C}^{\mathcal{RS}^s_{\mathcal{DB}}{}^*}_{\mathcal{S}}$, we conclude that $\mathcal{S}' \cap \mathcal{C}^*_{\mathcal{DB}} = \emptyset$. Using Proposition 5.1, we can therefore see that $\mathcal{R}_a \cup \mathcal{S}' \models \Sigma$.

If $\mathcal{R}_a \cup \mathcal{S}' \notin rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$, then there must exist some $\mathcal{R}'$ consistent with Σ such that $\mathcal{R}' <_{\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S}} \mathcal{R}_a \cup \mathcal{S}'$. We can write $\mathcal{R}' = \mathcal{R}'_a \cup \mathcal{R}'_s$, where $\mathcal{R}'_a = \mathcal{R}' \cap \mathcal{C}^*_{\mathcal{DB}}$ and $\mathcal{R}'_s = \mathcal{R}_{as} \setminus \mathcal{C}^*_{\mathcal{DB}}$. By Proposition 5.2, $\mathcal{R}_a \subseteq \mathcal{C}^*_{\mathcal{DB}}$. From Property (‡) for $\mathcal{R} = \mathcal{C}^*_{\mathcal{DB}}$, it thus follows that either $\mathcal{R}'_a <_{\mathcal{A}^*_{\mathcal{DB}}} \mathcal{R}_a$ or $\mathcal{R}'_s <_{\mathcal{S}} \mathcal{S}'$. Furthermore, by Proposition 5.1, $\mathcal{R}'_a \models \Sigma^a_{\mathcal{DB}}$ and $\mathcal{R}'_s \models \Sigma^s_{\mathcal{DB}}$. However, this contradicts that both $\mathcal{R}_a \in rep_{\mathcal{RS}^a_{\mathcal{DB}}}(\mathcal{A}^*_{\mathcal{DB}})$ and $\mathcal{S}' \in rep_{\mathcal{RS}^s_{\mathcal{DB}}}(\mathcal{S})$ hold. This proves $R_a \cup \mathcal{S}' \in rep_{\mathcal{RS}}(\mathcal{A}^*_{\mathcal{DB}} \cup \mathcal{S})$. □

Armed with the above concepts and results, we now turn to a data integration setting $\mathcal{I}$ 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 $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$.

Theorem 5.4 (Main) *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Then,*

1. *for every $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, there exists some $\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ such that $\mathcal{R} = \mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$;*
2. *for every $\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$, there exists some $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ such that $\mathcal{R} = \mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.*

Proof.

1. Recall that $ret(\mathcal{I}, \mathcal{D}) = \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ and that $rep_{\mathcal{I}}(\mathcal{D})$ coincides with $rep_{\mathcal{G}}(ret(\mathcal{I}, \mathcal{D}))$. Thus, by applying first Item 1 of Lemma 5.3 for $\mathcal{S} = \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$ and then Item 2 for $\mathcal{S} = \emptyset$, we obtain that for every $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, there exists some $\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ of form $\mathcal{R}' = (\mathcal{R} \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}'$, where $\mathcal{S}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} = \emptyset$. Hence, $\mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} = \mathcal{R} \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$. By Proposition 5.2, every $\mathcal{R} \in rep_{\mathcal{I}}(D)$ is of form $\mathcal{R} = (\mathcal{R} \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$. Therefore, $\mathcal{R} = (\mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.

2. Similarly, applying first Item 1 of Lemma 5.3 for $S = \emptyset$ and then Item 2 for $\mathcal{S} = \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$, we obtain that for every $\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$, there exists some $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ such that $\mathcal{R} = (\mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}'$, where $\mathcal{S}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} = \emptyset$. Furthermore, Proposition 5.2 implies $\mathcal{S}' = \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$. This concludes the proof. □

As a consequence, for computing the repairs of the retrieved global database, we can exploit the program Π_Σ that is part of the logic specification for querying $\mathcal{I}$ with a query Q (Definition 4.1), and proceed as follows:

(1) evaluate the program Π_Σ on $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$, and use the correspondence

$$rep_\mathcal{G}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}) \rightleftharpoons \mathrm{SM}(\Pi_\Sigma[\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}])$$

to obtain the repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$;

(2) intersect each repair obtained with $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$; and

(3) take for each such set the union with $\mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.

A drawback of this approach is that in Step (1), many facts outside $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ might be included in a repair of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$, which are stripped off subsequently in Step (2). Consider, for example, a global schema $\mathcal{G}$ containing the constraint $p(a)$, which forces every database instance for $\mathcal{G}$ to contain the fact $p(a)$, and the database $\mathcal{DB} = \{p(a)\}$ for $\mathcal{G}$. In this case $\mathcal{C}^*_{\mathcal{DB}} = \mathcal{C}_{\mathcal{DB}} = \mathcal{A}^*_{\mathcal{DB}} = \emptyset$ and $\mathcal{S}^*_{\mathcal{DB}} = \mathcal{DB}$. However, $rep_\mathcal{G}(\mathcal{A}^*_{\mathcal{DB}}) = \{\{p(a)\}\}$ and $\{p(a)\} \cap \mathcal{C}^*_{\mathcal{DB}} = \emptyset$.

5.2.1 Constraints $\mathbf{C}_1$

As we show in the following, the above problem does not hold as soon as we consider only integrity constraints that belong to the class $\mathbf{C}_1$, which has been introduced in Section 2.2. Notice that this implies that a constraint does not unconditionally enforce the inclusion of a fact in every database instance. Therefore, $\mathbf{C}_1$ is the class which contains all constraints that are usually issued on a database schema.

Proposition 5.5 *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$ such that $\Sigma \subseteq \mathbf{C}_1$, and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Then, each repair $\mathcal{R}'$ of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. $\mathcal{G}$ satisfies $\mathcal{R}' \subseteq \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$.*

Proof. By Item 1 of Lemma 5.3, each $\mathcal{R} \in rep_\mathcal{G}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ gives rise to a repair $\mathcal{R}' = \mathcal{R} \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. $\mathcal{RS}^a_{\mathcal{DB}} = \langle \Psi, \Sigma^a_{\mathcal{DB}} \rangle$. By Item 2 of Lemma 5.3, $\mathcal{R}'$ in turn gives rise to a repair $\mathcal{R}'' \in rep_\mathcal{G}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ of the form $\mathcal{R}' \cup \mathcal{S}'$ such that $\mathcal{S}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} = \emptyset$. In fact, by of Lemma 5.3, $\mathcal{S}'$ is any repair of $\mathcal{S} = \emptyset$ w.r.t. $\langle \Psi, \Sigma^s_{\mathcal{DB}} \rangle$. Since each constraint in $\Sigma^s_{\mathcal{DB}}$ has a nonempty body, it follows by the Property ($\star$) that $\mathcal{S}' = \emptyset$. Hence $\mathcal{R}'' = \mathcal{R} \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ is a repair of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. $\mathcal{G}$. Now if $\mathcal{R} \not\subseteq \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ would hold, then $\triangle(\mathcal{R}'', \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}) \subset \triangle(\mathcal{R}, \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ would hold, which by Property ($\star$) implies $\mathcal{R}'' <_{ret(\mathcal{I},\mathcal{D})} \mathcal{R}$. This is a contradiction. □

The proposition above allows us to exploit Theorem 5.4 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 $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. $\mathcal{G}$.

Corollary 5.6 *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$ such that $\Sigma \subseteq \mathbf{C}_1$. Then, for every source database $\mathcal{D}$ for $\mathcal{I}$, there exists a bijection $\mu : rep_\mathcal{I}(\mathcal{D}) \to rep_\mathcal{G}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$, such that for every $\mathcal{R} \in rep_\mathcal{I}(\mathcal{D})$, $\mathcal{R} = \mu(\mathcal{R}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.*

Proof. By Theorem 5.4, we know that:

1. $\forall \mathcal{R} \in rep_\mathcal{I}(\mathcal{D})$, $\exists \mathcal{R}' \in rep_\mathcal{G}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ such that $\mathcal{R} = \mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$;

2. $\forall \mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$, $\exists \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$ such that $\mathcal{R} = \mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.

The result follows by applying Proposition 5.5, and noting that $\mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} = \mathcal{R}'$. □

According to this result, repairs of the retrieved global database can be computed by avoiding step (2) of the procedure given above. Notice however that $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ has to be computed to single out both $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ and $\mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$.

5.2.2 Constraints $\mathbf{C}_0$

We next consider the more restrictive class $\mathbf{C}_0$, where constraints have only built-in relations in the head. Notably, the repairs of data integration systems with integrity constraints belonging to this class can be computed by focusing on the immediate conflicts in the database, without the need of computing the conflict closure set, which may be in general onerous. We will formally prove these properties in the rest of this section, starting by the following proposition.

Proposition 5.7 *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$ such that $\Sigma \subseteq \mathbf{C}_0$. Then, for every source database $\mathcal{D}$ for $\mathcal{I}$,*

1. *$\mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I},\mathcal{D})$;*
2. *each $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ satisfies $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq \mathcal{R} \subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ and $\triangle(\mathcal{R}, \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}) \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$;*
3. *each $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ satisfies $\mathcal{R} \cap \mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$;*
4. *each $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ satisfies $\mathcal{R} \cup (\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$.*

Proof.

1. By definition, $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ is the set of ground facts occurring in the ground instances of constraints in Σ that are violated in $ret(\mathcal{I},\mathcal{D})$. Since each of them is of the form $\bigwedge_{i=1}^{l} A_i(\vec{c}_i) \supset \bigvee_{k=1}^{n} \phi_k(\vec{d}_k)$, they might be violated only if all the body facts belong to $ret(\mathcal{I},\mathcal{D})$. That is, $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I},\mathcal{D})$.

2. Let $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$. That $\mathcal{R} \subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ follows from Item 1: Since each $\sigma \in \Sigma$ is of the form (2) with $m = 0$ and $l > 0$, "repairs" of ground constraints delete only tuples. Indeed, suppose that $\mathcal{R} \not\subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ and consider $\mathcal{R}' = \mathcal{R} \cap \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$. Then, $\mathcal{R}' \models \Sigma$ as well, and by Property ($\star$) we would arrive at a contradiction. Thus, we conclude $\mathcal{R} \subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$.

We next show that $\triangle(\mathcal{R}, \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}) \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$. Assume towards a contradiction that this does not hold. Since $\mathcal{R} \subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$, this implies that there exists some $p(t) \in \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{R}$ such that $p(t) \notin \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$. By minimality of $\mathcal{R}$, $p(t)$ occurs in the body of at least one constraint in $ground(\Sigma)$ of the form $\bigwedge_{i=1}^{l} a_i \supset \bigvee_{k=1}^{n} \phi_k$. No such constraint, however, is violated in $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$. Hence, $\mathcal{R} \cup \{p(t)\} \models \Sigma$, which by Property ($\star$) implies that $\mathcal{R} \notin rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$; this is a contradiction. Therefore, $\triangle(\mathcal{R}, \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}) \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ holds. From this and $\mathcal{R} \subseteq \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ it follows that $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq \mathcal{R}$.

3. Consider $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$. Towards a contradiction, suppose $\mathcal{R} \cap \mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \notin rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$. Then,some $\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ exists such that $\mathcal{R}' <_{\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}} \mathcal{R}$. Since all constraints have only built-ins in their heads, $\mathcal{R}' \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$. But then $(\mathcal{R} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{R}' <_{\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}} \mathcal{R}$ contradicts that $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$.

4. For every $\mathcal{R}' \in rep(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ w.r.t. $\mathcal{G}$, it holds that $\mathcal{R} = (\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{R}' \models \Sigma$. By Item 2 and Property ($\star$), it thus follows that no $\mathcal{R}'' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ exists such that $\mathcal{R}'' <_{\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}} \mathcal{R}$. Hence, $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$. □

We are now ready to prove that, under constraints belonging to the class $\mathbf{C}_0$, $rep_{\mathcal{I}}(\mathcal{D})$ can be obtained by simply computing the repairs of the conflicting facts, $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$, in place of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$, thus avoiding the construction of $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$.

Theorem 5.8 *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$ such that $\Sigma \subseteq \mathbf{C}_0$. Then, for each source database $\mathcal{D}$ for $\mathcal{I}$, there exists a bijection $\nu : rep_{\mathcal{I}}(\mathcal{D}) \rightarrow rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$, such that for each $\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, $\mathcal{R} = \nu(\mathcal{R}) \cup (ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$.*

Proof. By Corollary 5.6, we have a bijection $\mu : rep_{\mathcal{I}}(\mathcal{D}) \rightarrow rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ such that the repairs of $\mathcal{I}$ w.r.t. $\mathcal{D}$ are given by $\mu(\mathcal{R}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$, for all $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$. Items 1 and 3 of Proposition 5.7 and the fact that each repair $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ satisfies $\mathcal{R} \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ (apply Proposition 5.7 for $\mathcal{DB} = \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$), imply that all repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ are given by $(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{R}$, where $\mathcal{R} \in rep(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$. Hence, the mapping $\nu : rep_{\mathcal{I}}(\mathcal{D}) \rightarrow rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ given by $\nu(\mathcal{R}) = \mu(\mathcal{R}) \cap \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ is a bijection such that

$$\begin{aligned}
\mathcal{R} &= \mu(\mathcal{R}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})} \\
&= \nu(\mathcal{R}) \cup (\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})} \\
&= \nu(\mathcal{R}) \cup ((ret(\mathcal{I},\mathcal{D}) \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}) \cup (ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}) \\
&= \nu(\mathcal{R}) \cup (ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})})
\end{aligned}$$

□

As a consequence, for computing the repairs of the retrieved global database in the above setting, we can proceed as follows:

(1) evaluate the program Π_Σ on $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$, in order to obtain repairs of $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$; and

(2) take for each such repair the union with $ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$.

We finally point out that, since $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I},\mathcal{D})$, the computation of the set $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ can efficiently be carried out by means of suitable SQL statements.

Example 5.4 Recall that $\mathcal{C}_{ret(\mathcal{I}_0,\mathcal{D}_0)} = \{team(RM, \textit{Roma}, 10), team(RM, \textit{Real Madrid}, 10)\}$ (see Example 5.2). The two repairs of $\mathcal{C}_{ret(\mathcal{I}_0,\mathcal{D}_0)}$ are $\mathcal{R}''_1 = \{team(RM, \textit{Roma}, 10)\}$ and $\mathcal{R}''_2 = \{team(RM, \textit{Real Madrid}, 10)\}$.

According to Theorem 5.8, these two repairs are sufficient for computing the repair of the retrieved global database $ret(\mathcal{I}_0,\mathcal{D}_0)$. Indeed, noticing that $ret(\mathcal{I}_0,\mathcal{D}_0) \setminus \mathcal{C}_{ret(\mathcal{I}_0,\mathcal{D}_0)} = \{coach(7, \textit{Camacho}, RM), player(10, \textit{Totti}, RM), player(9, \textit{Beckham}, MU)\}$, it is easy to see that $\mathcal{R}_1 = \mathcal{R}''_1 \cup ret(\mathcal{I}_0,\mathcal{D}_0) \setminus \mathcal{C}_{ret(\mathcal{I}_0,\mathcal{D}_0)}$ and $\mathcal{R}_2 = \mathcal{R}''_2 \cup ret(\mathcal{I}_0,\mathcal{D}_0) \setminus \mathcal{C}_{ret(\mathcal{I}_0,\mathcal{D}_0)}$, where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the only two repairs of $ret(\mathcal{I}_0,\mathcal{D}_0)$, as evidenced in Example 3.1. □

5.2.3 Factorization

The above decomposition approach can be further refined by factorizing repairs into independent components, whose orthogonal combinations yield all repairs. This may be achieved by partitioning the affected

part $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ of the retrieved global database $ret(\mathcal{I},\mathcal{D})$ for source database $\mathcal{D}$ w.r.t. $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ suitably into disjoint subparts $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}{}^{(1)}, \ldots, \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}{}^{(m)}$, such that the repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ w.r.t. the global schema $\mathcal{G}$ are obtained by combining the repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}{}^{(1)}$, $\ldots$, $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}{}^{(m)}$ w.r.t. $\mathcal{G}$ in all possible ways. Thus, intuitively independent constraint violations (e.g., different violations of a key constraint) may be independently repaired and combined. Furthermore, factorization leads to an exponential saving in storing repairs in such a case.

We describe here such a partitioning for $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ where the constraints in $\mathcal{G} = \langle \Psi, \Sigma \rangle$ have nonempty bodies. Let us call a partitioning $\mathcal{C}_1 \ldots, \mathcal{C}_m \subseteq \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ of $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ *repair-compliant*, if every pair of facts $p(t)$, $p'(t')$ which are constraint-bounded in $\mathcal{G}$ belongs to the same component $\mathcal{C}_i$. Then, we can factorize repairing $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ into repairing the disjoint parts of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ on $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Intuitively, all repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}{}^{(i)} = \mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \cap \mathcal{C}_i$ are confined to $\mathcal{C}_i$, and by the abstract properties ($\star$), (†), and (‡) of the preference ordering, they can be orthogonally combined with the repairs of all other parts $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \cap \mathcal{C}_j$.

More formally, the following result can be shown with similar arguments as in the proofs of Lemma 5.3 and Theorem 5.4.

Theorem 5.9 (Factorization) *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system, where $\mathcal{G} = \langle \Psi, \Sigma \rangle$ and $\Sigma \subseteq \mathbf{C}_1$ and let $\mathcal{D}$ be a source database for $\mathcal{I}$. Suppose that $\mathcal{C}_1 \ldots, \mathcal{C}_m$ is a repair-compliant partitioning of $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$. Then,*

$$rep_{\mathcal{I}}(\mathcal{D}) = \{\mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_m \mid \mathcal{R}_i \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})} \cap \mathcal{C}_i), 1 \leq i \leq m\}.$$

For the more restrictive yet practically important class $\mathbf{C}_0$ a similar factorization result holds with repair-compliant partitioning $\mathcal{C}_1, \ldots, \mathcal{C}_m$ of $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ in place of the larger $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$.

A repair-compliant partitioning $\mathcal{C}_1 \ldots, \mathcal{C}_m$ of $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ (resp., $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$) can be efficiently computed, using techniques for computing the connected components of a graph. Note that $\mathcal{C}_i$ is a union of connected components of the graph with nodes in $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$ (resp., $\mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$) and edges between each pair of facts which are constraint-bounded in $\mathcal{G}$.

5.3 Recombination

Let us now turn to the evaluation of a user query Q issued over a data integration system $\mathcal{I}$, i.e., to computing its consistent answers w.r.t. a source database $\mathcal{D}$ for $\mathcal{I}$. According to the definition given in Section 3.1, a tuple t is a consistent answer to Q if t is in the answer to Q on any repair of the retrieved global database $ret(\mathcal{I},\mathcal{D})$. Then, in principle, we need to compute each such repair. To this aim, we can resort to repairing the set $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$, as established in the decomposition step. The following theorem indicates how to recombine the repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ with $\mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$ in order to provide consistent answers to Q.

Theorem 5.10 *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,*

$$ans(Q,\mathcal{I},\mathcal{D}) = \bigcap_{\mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})} Q[(\mathcal{R}' \cap C^*_{ret(\mathcal{I},\mathcal{D})}) \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}] \tag{3}$$

Proof. Recall that, by definition, $ans(Q,\mathcal{I},\mathcal{D}) = \{t \mid t \in Q[\mathcal{R}] \text{ for each } \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})\} = \bigcap_{\mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})} Q[\mathcal{R}]$. Then, the thesis follows by applying Theorem 5.4, stating that $\forall \mathcal{R} \in rep_{\mathcal{I}}(\mathcal{D})$, $\exists \mathcal{R}' \in rep_{\mathcal{G}}(\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})})$ such that $\mathcal{R} = \mathcal{R}' \cap \mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})} \cup \mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$. □

Note that the number of repairs of $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ is in general exponential in the number of violated constraints, and hence efficient computation of the intersection in (3) 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.

6 Repair Compilation

In this section, we describe some practical recombination strategies, for effectively exploiting currently available answer set solvers, such as DLV or Smodels, in order to efficiently query data integration systems. Then, based on such strategies, we devise a general architecture (showed in Figure 6) that properly interleaves the computational power of answer set solvers (for reasoning on the portion of data that really needs to be repaired) and the scalability of database systems (for implementing the recombination step in a way which circumvents the evaluation of the query Q on each repair of $ret(\mathcal{I},\mathcal{D})$ separately). The architecture shows that our approach can be profitably implemented on the top of any available answer set engine thereby significantly speeding up query answering in data integration systems.

For the sake of simplicity, we deal here with constraints from the class $\mathbf{C}_0$. In this case, according to Theorem 5.8, there exists a bijection between $rep_{\mathcal{I}}(\mathcal{D})$ and $rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$, and for each $\mathcal{R} \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$ we have (by Proposition 5.7) that $\mathcal{R} \subseteq \mathcal{C}_{ret(\mathcal{I},\mathcal{D})} \subseteq ret(\mathcal{I},\mathcal{D})$. In other words, equation 3 in Theorem 5.10 can be rewritten to

$$ans(Q,\mathcal{I},\mathcal{D}) = \bigcap_{\mathcal{R}\in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})} Q[\mathcal{R} \cup (ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})})].$$

We point out that our recombination technique also applies in the presence of constraints of general form. In such a case, the source of complexity lies in the computation of $\mathcal{C}^*_{ret(\mathcal{I},\mathcal{D})}$.

For ease of exposition, in the following we denote with $\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$ the set $ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$ and with $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}$ the set $ret(\mathcal{I},\mathcal{D}) \setminus \mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$. Since, for constraints of class $\mathbf{C}_0$, $\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$ and $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}$ play the same role that $\mathcal{S}^*_{ret(\mathcal{I},\mathcal{D})}$ and $\mathcal{A}^*_{ret(\mathcal{I},\mathcal{D})}$ play for general integrity constraints, we simply refer to them as the safe and the affected portion of $ret(\mathcal{I},\mathcal{D})$ respectively. Notice also that $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})} = \mathcal{C}_{ret(\mathcal{I},\mathcal{D})}$, hence, equation 3 in Theorem 5.10 can be further rewritten to

$$ans(Q,\mathcal{I},\mathcal{D}) = \bigcap_{\mathcal{R}\in rep_{\mathcal{G}}(\mathcal{A}_{ret(\mathcal{I},\mathcal{D})})} Q[\mathcal{R} \cup \mathcal{S}_{ret(\mathcal{I},\mathcal{D})}].$$

6.1 Marking the Retrieved Global Database

The basic idea of our approach is to encode all repairs of the retrieved global database into a single database over which the query can be evaluated by means of standard database techniques. More precisely, for each global relation predicate s, we construct a new relation predicate s_m by adding the auxiliary attribute *mark* to the attributes in s. Values for the mark attribute are strings of bits, each assuming either the value 0 or 1. To each tuple $t \in s^{ret(\mathcal{I},\mathcal{D})}$, we associate a mark $'b_1 \ldots b'_n$ such that, for every $i \in \{1,\ldots,n\}$, $b_i = 1$ if t belongs to the i-th repair $\mathcal{R}_i \in rep_{\mathcal{G}}(\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}) = \{\mathcal{R}_1,\ldots,\mathcal{R}_n\}$, $b_i = 0$ otherwise (indexing the repairs is easy, e.g. using the order in which the deductive database system computes them). The resulting tuple is stored in the corresponding relation predicate s_m. The extension of all such s_m constitutes the *marked* database, denoted by $\mathcal{M}_{ret(\mathcal{I},\mathcal{D})}$. Note that the facts in $\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$ (the bulk of data) can be marked without any

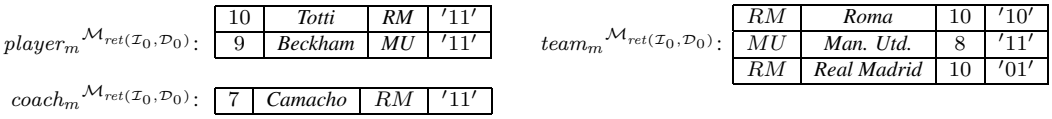


Figure 4: The retrieved global database of our running example after marking.

preprocessing, as they belong to every repair $\mathcal{R}_i$; hence, their mark is $'11\ldots1'$. For our running example, the marked database derived from the repairs in Figure 2 is shown in Figure 4.

6.2 Query Reformulation

We now show how a non-recursive Datalog$^\neg$ query Q over a data integration system $\mathcal{I}$ can be reformulated into an SQL query whose evaluation over the marked database obtained from $ret(\mathcal{I}, \mathcal{D})$ returns the consistent answers to Q w.r.t. $\mathcal{D}$. Let $r : h(\vec{x}) \leftarrow B(\vec{x})$ be a safe rule of form

$$p_0(\vec{x}_0) \leftarrow p_1(\vec{x}_1), ..., p_l(\vec{x}_l),\ not\ p_{l+1}(\vec{x}_{l+1}), ...,\ not\ p_{l+k}(\vec{x}_{l+k})^2. \tag{4}$$

Let $t_{i,j}$ denote the j-th term in $p_i(\vec{x}_i) = p_i(t_{i,1}, \ldots, t_{i,k_i})$, where $0 \leq i \leq l + k$ and $1 \leq j \leq k_i$. We associate with r a normalized rule r' obtained from it as follows:

1. Replace each $t_{i,j}$ by a new variable $y_{i,j}$.
2. if $t_{i,j}$ is a constant c, then add the equality atom $y_{i,j} = c$ to the body;
3. if $t_{i,j}$ is a variable x, then add the equality atom $y_{i,j} = y_{i',j'}$ to the body, where $t_{i',j'}$ is the first occurrence of x in the body of r (from left to right), except for $i = i'$ and $j = j'$. (Note that safety of r guarantees $0 \leq i' \leq l$.)

In query reformulation, we furthermore use the following functions `ANDBIT`, `INVBIT`, and `SUMBIT` (which can be easily defined in a relational DBMS):

- `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 $'c_1 \ldots c'_n$, where $c_i = a_i \wedge b_i$ is the Boolean "and," $i = 1, \ldots, n$;
- `INVBIT` is a unary function that takes as its input a bit string $'a_1 \ldots a'_n$ and returns $'c_1 \ldots c'_n$, where $c_i = \neg a_i$ is the Boolean complement, $i = 1, \ldots, n$;
- `SUMBIT` is an aggregate function such that given m strings of form $'b_{i,1} \ldots b'_{i,n}$, $i = 1, \ldots, m$, it returns $'c_1 \ldots c'_n$, where $c_j = b_{1,j} \vee \ldots \vee b_{m,j}$ is the Boolean "or," $j = 1, \ldots, n$.

Let $Q = \langle q, \mathcal{P} \rangle$ be a query of arity n, where $\mathcal{P}$ consists of normalized rules $r : h(\vec{x}') \leftarrow B(\vec{y}'), e(\vec{z})$, where $e(\vec{z})$ are all the equality atoms introduced in normalization. Let a_i, $1 \leq i \leq n$, be pairwise distinct identifiers for the attributes of a predicate of arity n. Then, each r is translated into the following SQL statement SQL_r (notice that, in the statements below, each relation symbol p_i occurring in r is transformed in the corresponding marked symbol p_{i_m}):

$\texttt{SELECT}\ p_{i'm}.a_{j'}\ \texttt{AS}\ a_j$ (*for each atom* $y_{0,j} = y_{i',j'}$ *in* $e(\vec{z})$)
$c\ \texttt{AS}\ a_j,$ (*for each atom* $y_{0,j} = c$ *in* $e(\vec{z})$)
$(p_{1m}.mark\ \texttt{ANDBIT} \ldots \texttt{ANDBIT}\ p_{lm}.mark\ \texttt{ANDBIT}\ \texttt{INVBIT}(n\_p_{l+1m}.mark)\ \texttt{ANDBIT}$
$\ldots\ \texttt{ANDBIT}\ \texttt{INVBIT}(n\_p_{l+km}.mark))\ \texttt{AS}\ mark$
$\texttt{FROM}\ p_{1m}, \ldots, p_{lm}, SQL_{r,l+1}, \ldots, SQL_{r,l+k}$
$\texttt{WHERE}\ p_{im}.a_j{=}p_{i'm}.a_{j'},$ (*for each atom* $y_{i,j} = y_{i',j'}$ *in* $e(\vec{z}), 0 < i \leq l$)
$p_{im}.a_j{=}c,$ (*for each atom* $y_{i,j} = c$ *in* $e(\vec{z}), 0 < i \leq l$)
$n\_p_{im}.a_j{=}p_{i'm}.a_{j'},$ (*for each atom* $y_{i,j} = y_{i',j'}$ *in* $e(\vec{z}), l < i$)
$n\_p_{im}.a_j{=}c$ (*for each atom* $y_{i,j} = c$ *in* $e(\vec{z}), l < i$).

where each $SQL_{r,h}$, $l < h \leq l + k$, is a subquery of form:

($\texttt{SELECT}\ *\ \texttt{FROM}\ p_{hm}$
$\texttt{UNION}$
$\texttt{SELECT}\ p_{i'm}.a_{j'}\ \texttt{AS}\ a_h,$ (*for each atom* $y_{h,j} = y_{i',j'}$ *in* $e(\vec{z})$)
$c\ \texttt{AS}\ a_h,$ (*for each atom* $y_{h,j} = c$ *in* $e(\vec{z})$)
$'0 \ldots 0'\ \texttt{AS}\ mark$
$\texttt{FROM}\ p_{1m}, \ldots, p_{lm}$
$\texttt{WHERE}\ p_{im}.a_j{=}p_{i'm}.a_{j'},$ (*for each atom* $y_{i,j} = y_{i',j'}$ *in* $e(\vec{z}), 0 < i \leq l$)
$p_{im}.a_j{=}c,$ (*for each atom* $y_{i,j} = c$ *in* $e(\vec{z}), 0 < i \leq l$)
$\texttt{ROW}(a_1, \ldots, a_{k_i})\ \texttt{NOT IN}\ (\texttt{SELECT}\ a_1, \ldots, a_{k_i}\ \texttt{FROM}\ p_{im})$
) $\texttt{AS}\ n\_p_{hm}$.

Roughly speaking, in the statement SQL_r, the ANDBIT operator allows us to obtain the mark $'b_1, \ldots, b'_n$ of each tuple t computed for the relation predicate h, according to rule r. More precisely, for $i \in \{1, \ldots, n\}$, $b_i = 1$ if t is in the repair $\mathcal{R}_i \in rep_{\mathcal{G}}(\mathcal{C}_{ret(\mathcal{I},\mathcal{D})})$, $b_i = 0$ otherwise.

For each negative literal $not\ p_{hm}(\vec{y}_h)$, the marks must be inverted, where missing tuples (which do not belong to any repair, and thus would be marked $'0 \ldots 0'$) must be taken into account. To this aim, $SQL_{r,h}$ singles out the tuples returned by the positive body of the rule r, projects them on the attributes that are in join with the attributes in p_{h_m}, and returns, with mark $'0 \ldots 0'$, those of such tuples that do not occur in p_{hm} (taking then the union with the tuples in p_{hm} itself). The operator INVBIT guarantees that, for each such tuple, the mark returned by SQL_r is the one computed in the positive part of the query (in these cases indeed the negative body is satisfied in any repair). Notice that safety of the rule r ensures that the two queries in $SQL_{r,h}$ have the same arity.

All rules, $r_1, \ldots, r_\ell$, defining the same predicate h of arity n, are collected into a view by the SQL statement SQL_h:

$\texttt{CREATE VIEW}\ \ h_m(a_1, \ldots, a_n, mark)\ \ \texttt{AS}$
$\texttt{SELECT}\ a_1, \ldots, a_n, \texttt{SUMBIT}(mark)$
$\texttt{FROM}\ (SQL_{r_1}\ \texttt{UNION} \ldots \texttt{UNION}\ SQL_{r_\ell})$
$\texttt{GROUP BY}\ a_1, \ldots a_n.$

Finally, the answers to the query $Q = \langle q, \mathcal{P} \rangle$ are obtained through the statement SQL_Q:

$\texttt{SELECT}\ a_1, \ldots a_n\ \texttt{FROM}\ q_m\ \texttt{WHERE}\ mark =' 1 \ldots 1',$

where q_m is the view predicate defined by the statement SQL_q.

The above statement computes the query answers by considering only the facts that evaluate to true in all repairs. Omitting a proof, we state the following correctness result.

Proposition 6.1 *Let $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ be a data integration system , $\mathcal{D}$ a source database for $\mathcal{I}$, and Q a query over $\mathcal{G}$. Then, $ans(Q, \mathcal{I}, \mathcal{D})$ is the set of tuples computed by SQL_Q on the marked database $\mathcal{M}_{ret(\mathcal{I},\mathcal{D})}$.*

Example 6.1 The query in our running query has two rules, viz.

$$r_1 : q(x) \leftarrow player(x, y, z) \text{ and } r_2 : q(x) \leftarrow team(v, w, x).$$

Their normalized versions are:

$$r'_1 : q(y_{0,1}) \leftarrow player(y_{1,1}, y_{1,2}, y_{1,3}), y_{0,1} = y_{1,1};$$
$$r'_2 : q(y_{0,1}) \leftarrow team(y_{1,1}, y_{1,2}, y_{1,3}), y_{0,1} = y_{1,3}.$$

Thus, they translate into corresponding SQL statements SQL_{r_1} and SQL_{r_2}:

```
SELECT player_m.Pcode AS a_1,
       player_m.mark AS mark,
FROM player_m;
```

```
SELECT team_m.Tleader AS a_1,
       team_m.mark AS mark,
FROM team_m;
```

Finally, a view for the query predicate q and the final query SQL_Q are expressed as:

```
CREATE VIEW q_m(a_1, mark) AS
       SELECT a_1, SUMBIT(mark)
       FROM (SQL_{r_1} UNION SQL_{r_2})
       GROUP BY a_1;

SELECT a_1 FROM q_m WHERE mark =' 11';
```

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

Example 6.2 Let us now consider a query $Q = \langle q, \mathcal{P} \rangle$ asking for players that are not team leaders. Here $\mathcal{P}$ contains again two rules, of which one defines an auxiliary predicate $leader$:

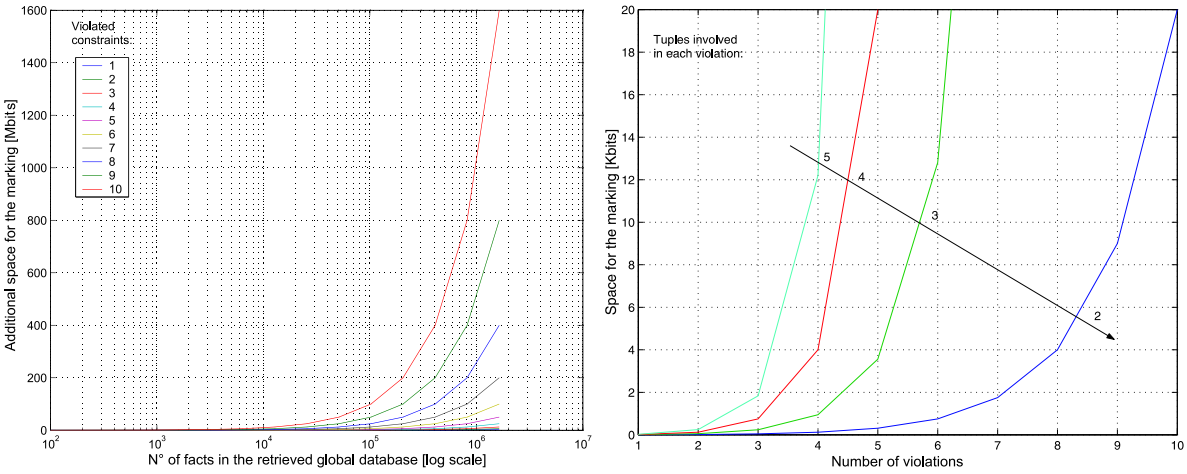
$$r_1 : q(x) \leftarrow player(x, y, z),\ not\ leader(x);$$
$$r_2 : leader(x) \leftarrow team(v, w, x).$$

The use of negation is reflected in $SQL_{r'_1}$ (let r'_1, r'_2 be the normalized versions of r_1, r_2):

```
SELECT player_m.Pcode AS a_1,
       (player_m.mark ANDBIT INVBIT(n_leader_m.mark)) AS mark,
FROM player_m,
     (SELECT player_m.Pcode AS a_1, '00' AS mark
      FROM player_m WHERE ROW(player_m.Pcode) NOT IN (SELECT a_1 FROM leader_m)
      UNION SELECT * FROM leader_m) AS n_leader_m
WHERE n_leader_m.a_1 = player_m.Pcode;
```

Whereas the use of an auxiliary predicate causes the creation of two views: one for each intensional predicate. The respective SQL statements SQL_q and SQL_{leader}, resemble the statement SQL_q of the previous example, however, each of them just depends on a single SQL query ($SQL_{r'_1}$ and $SQL_{r'_2}$, respectively). Moreover, it is easy to see that $SQL_{r'_2}$ and SQL_Q equal the corresponding queries in the previous example. Hence, as is easily retraced, the answer to query Q consists of the code *9*, as expected. □

Clearly, the SQL statements SQL_r, SQL_h, and SQL_Q can by optimized (which will be done by the DBMS anyway), and we do not consider optimization here. We remark that the final query, SQL_Q, could be as well integrated into the view definition, SQL_q, for the query predicate q. By keeping the query definition SQL_Q separate, however, other query semantics can easily be expressed; e.g., possibilistic query semantics, which selects those tuples which are computed by the query with respect to at least one repair, is obtained by replacing the condition in the `WHERE` clause by $\mathbf{mark} \neq \text{'}0 \ldots 0\text{'}$. We finally remark that the reformulation technique is amenable to other semantics of negation in queries as well. In particular, if negation is evaluated over *all* repairs, this can also be accomplished with slight modifications.



(a) (b)

Figure 5: Additional space for marking w.r.t. the size of the global database $ret(\mathcal{I}, \mathcal{D})$

6.3 Scaling the Technique

Since the number of repairs is exponential in the number of violated constraints, the marking string can be of considerable length. For instance, 10 constraint violations, involving two facts each, give rise to 2^{10} repairs, and hence 1 Kbit is needed for marking each tuple.

Figure 5.(a) shows the additional space needed in our running example, depending on the size of the retrieved global database, $ret(\mathcal{I}, \mathcal{D})$, for different numbers of constraint violations (assuming each violation has two conflicting facts). Notice, for instance, that a relatively small retrieved global database of 100,000 facts requires almost 12,5 MB additional space, most of it for marking the safe part.

Even if this may seem a serious limitation to the applicability of our technique, we next show that the space needed for marking the inconsistent database can be easily reduced in a considerable way. To this aim, we refine our technique in a way such that only the affected part of the retrieved global database needs actually to be marked. More specifically, to each global relation symbol r we associate two predicate symbols r_{safe} and r_{aff}, which are intended to store the tuples that occur in the safe and the affected portion of $r^{ret(\mathcal{I},\mathcal{D})}$, respectively. Also, we construct the database instance $\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}$ by replacing each relation symbol r in $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}$ with r_{aff}, and the database instance $\mathcal{S}'_{ret(\mathcal{I},\mathcal{D})}$ by replacing each relation symbol r in $\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$ with r_{safe}, i.e., we have that $r_{aff}^{\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}} = \{t \mid r(t) \in \mathcal{A}_{ret(\mathcal{I},\mathcal{D})}\}$ and $r_{safe}^{\mathcal{S}'_{ret(\mathcal{I},\mathcal{D})}} = \{t \mid r(t) \in \mathcal{S}_{ret(\mathcal{I},\mathcal{D})}\}$. Then, given a query $Q = \langle q, \mathcal{P}\rangle$ over $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M}\rangle$, where $\mathcal{G} = \langle \Psi, \Sigma\rangle$, and $\mathcal{P}$ is assumed normalized, we proceed as follows:

- for each rule $r : h(\vec{x}) \leftarrow B(\vec{x})$ of form (4) belonging to $\mathcal{P}$, we replace each atom $p_j(\vec{x}_j)$ of its positive body, i.e., $1 \leq j \leq l$, by $p_{aff_j}(\vec{x}_j) \vee p_{safe_j}(\vec{x}_j)$;
- we rewrite the resulting rule body into disjunctive normal form $B_1(\vec{x}) \vee \cdots \vee B_n(\vec{x})$;
- we replace in $B_i(\vec{x})$ each negative literal $not\ p_j(\vec{x}_j)$ with a global relation $p_j \in \Psi$ by the literals $not\ p_{aff_j}(\vec{x_j})$, $not\ p_{safe_j}(\vec{x}_j)$, and let $B'_i(\vec{x})$ be the result;
- we replace r with the rules $r_i : h(\vec{x}) \leftarrow B'_i(\vec{x})$, for $1 \leq i \leq n$;

– in the SQL statement SQL_{r_i} for r_i, replace every $p_{safe_j m}$ by p_{safe_j}, and $p_{safe_j}.mark$ by $'1 \ldots 1'$.

It is easy to see, that the SQL reformulation of the query Q refined as described above can be evaluated over the database $\mathcal{M}_{\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}} \cup \mathcal{S}'_{ret(\mathcal{I},\mathcal{D})}$. In other words, the reformulated query can be evaluated on a database instance in which only affected tuples have been marked.

Notice that the above rewriting is exponential in the size of Q (more precisely, in the number of atoms). However, as commonly agreed in the database community, the overhead in query complexity usually pays off the advantage gained in data complexity.

With this approach, the additional space depends only on the size of $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}$ but not on the size of $\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$. Figure 5.(b) shows the additional space for marking $\mathcal{A}_{ret(\mathcal{I},\mathcal{D})}$ depending on the number of violations. For example, for 10 constraint violations involving two tuples each, the required marking space is 2*10*2^{10} bits = 2.5 KB, independently on the size of $ret(\mathcal{I},\mathcal{D})$. Furthermore, by allotting 5 MB (=2*20*2^{20} bits) marking space, the technique may scale up to 20 constraint violations, involving two tuples each. The improvements w.r.t. to the naive implementation should be evident. However, for a large number of violated constraints, say 800, the technique becomes infeasible.

There are various possibilities to further increase scalability of the marking approach. Some of them are discussed in the following.

Sliced marking. Within a given reasonable marking space, any number n of violations can be handled by evaluating the query Q *incrementally* over a sequence of partially marked databases $\mathcal{M}^i_{ret(\mathcal{I},\mathcal{D})}$, $1 \leq i \leq \lceil n/m \rceil$, which contain the marks of m repairs at a time, i.e., each relation r in $\mathcal{M}^i_{ret(\mathcal{I},\mathcal{D})}$ is marked with the bits $'b_{m(i-1)+1} \cdots b'_{m*i}$, which are a small portion of the full marking $'b_1 \cdots b'_n$. Furthermore, we recall that by relevance pruning, the number of conflicts which actually need to be considered to answer a particular query Q might be drastically smaller than the total number of conflicts in the global database.

Factorization. Splitting up repairs by factorization as in Section 5.2.3 can help to drastically reduce the marking space. For the above example of 20 violated constraints, only a few bytes of marking space is needed for the 10 components $\mathcal{A}^{(1)}_{ret(\mathcal{I},\mathcal{D})}, \ldots \mathcal{A}^{(10)}_{ret(\mathcal{I},\mathcal{D})}$. in total.

The marking strategy can be extended to support dynamic combination of the marked databases $\mathcal{M}^{(1)}_{ret(\mathcal{I},\mathcal{D})}$, $\ldots$, $\mathcal{M}^{(m)}_{ret(\mathcal{I},\mathcal{D})}$, which store the set of repairs of $\mathcal{A}^{(1)}_{ret(\mathcal{I},\mathcal{D})}, \ldots \mathcal{A}^{(m)}_{ret(\mathcal{I},\mathcal{D})}$. To this end, the *mark* strings $'b_1 \ldots b'_n$ of tuples t in any relation r in $\mathcal{M}^{(i)}_{ret(\mathcal{I},\mathcal{D})}$ must be "exploded" in order to correctly reflect the membership of $r(t)$ in a combined repair. For example, if t_1 and t_2 are tuples in repairs of components $rep(\mathcal{A}^{(1)}_{ret(\mathcal{I},\mathcal{D})}) = \{\mathcal{R}_{1,1}, \mathcal{R}_{1,2}\}$, and $rep(\mathcal{A}^{(2)}_{ret(\mathcal{I},\mathcal{D})}) = \{\mathcal{R}_{2,j} \mid 1 \leq j \leq 4\}$ with markers $'10'$ and $'1001'$, respectively, then their exploded markers for the combined repairs may be $'1111000'$ and $'10011001'$, corresponding to lexicographic enumeration of $rep(\mathcal{A}^{(1)}_{ret(\mathcal{I},\mathcal{D})}) \times rep(\mathcal{A}^{(2)}_{ret(\mathcal{I},\mathcal{D})})$. We omit further details.

In combination with sliced marking, very large sets of repairs can be stored and recombined within manageable resources. We note that the marked databases $\mathcal{M}^{(i)}_{ret(\mathcal{I},\mathcal{D})}$ can be computed in parallel. Furthermore, in order to speed up online query answering, marked databases for a factorization of all conflicts in the retrieved global database $ret(\mathcal{I},\mathcal{D})$ can be precomputed at design time or after updates (if infrequent), and for query evaluation those marked databases accessed which are relevant to a query.

Core computations. For positive queries Q, the number of groups of repairs that need to be considered can be reduced in some cases by first computing a core of the query result by applying Q on the safe part

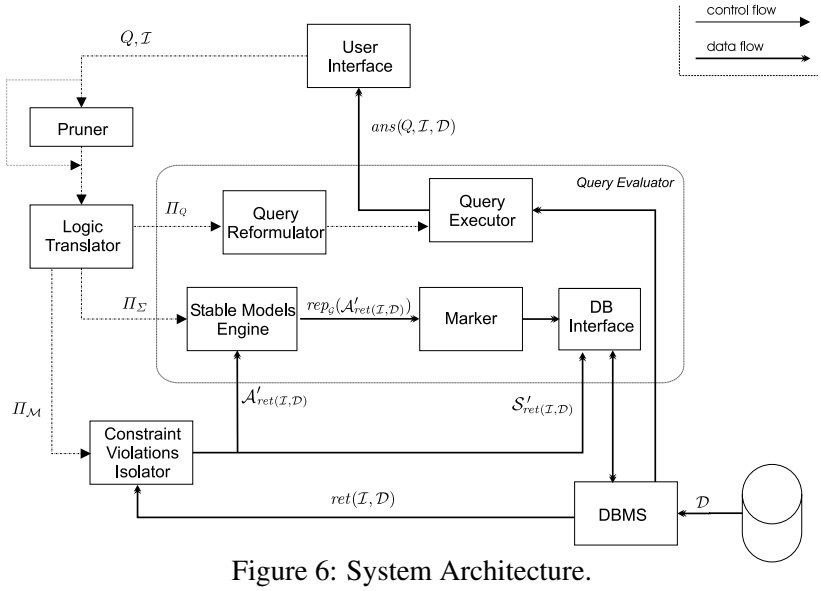


Figure 6: System Architecture.

$\mathcal{S}_{ret(\mathcal{I},\mathcal{D})}$ of the retrieved database, and stop the evaluation whenever the current query result after evaluation of $\mathcal{M}^i_{ret(\mathcal{I},\mathcal{D})}$ coincides with this core (this can be efficiently checked on the database using `COUNT DISTINCT` directives). Similar techniques for bounding query results from above and below have been discussed in [19], and for certain classes of queries this can lead to significant savings.

6.4 General Architecture

The general architecture supporting our repair compilation technique is shown in Figure 6. Its kernel is constituted by the *Query Evaluator*, which implements both the marking and the recombination step.

The functionalities of the components in the architecture are as follows.

- *Pruner:* It takes the user query Q and the specification of the data integration systems $\mathcal{I}$, and produces an equivalent specification (w.r.t. Q) in which relations and constraints not relevant for answering the query are stripped off.

- *Logic Translator:* It takes the specification of $\mathcal{I}$ relevant for Q returned by the Pruner, and produces the logic program $\Pi_\mathcal{I}(Q) = \Pi_\mathcal{M} \cup \Pi_\Sigma \cup \Pi_Q$. $\Pi_\mathcal{I}(Q)$ is subsequently processed by the Conflict Isolator module, which takes $\Pi_\mathcal{M}$ as input, the Stable Models Engine, which considers Π_Σ, and the Query Reformulator, whose input is Π_Q.

- *Constraint Violation Isolator:* It is responsible of processing the program $\Pi_\mathcal{M}$ by accessing the individual sources to compute the retrieved global database $ret(\mathcal{I}, \mathcal{D})$. Specifically, this module produces a set of SQL views corresponding to the (GAV) mapping in $\mathcal{I}$, which can be directly evaluated by a DBMS. Notice that, in the actual implementation of the module, SQL views are not aimed at simply materializing $ret(\mathcal{I}, \mathcal{D})$, but they produce $\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}$ and $\mathcal{S}'_{ret(\mathcal{I},\mathcal{D})}$, i.e., the safe and the affected databases in which each relation symbol r is replaced with r_{aff} or r_{safe}, respectively.

- *Stable Models Engine:* It takes the "affected" relations (constituting the database $\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}$) and computes the set of repairs $rep_\mathcal{G}(\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})})$ by exploiting the program Π_Σ.

- *Marker:* It wraps the output of the Stable Models Engine and produces the statements needed for storing the marked relations.

• *DB Interface:* It is responsible for the interfacing between the Stable Models Engine and the DBMS. Specifically, it executes the statements needed for storing the marked relations in order to implement the technique described in Section 6.3.

• *Query Reformulator:* It takes the user query and transforms it in a suitable set of SQL statements that can be executed directly over the DBMS.

• *Query Executor:* It is responsible for the execution of the reformulated query.

• *DBMS:* It is a Database Management System which stores data wrapped (not showed) from the sources and the marked database $\mathcal{M}_{\mathcal{A}'_{ret(\mathcal{I},\mathcal{D})}} \cup \mathcal{S}'_{ret(\mathcal{I},\mathcal{D})}$ over which the SQL reformulation of the query Q has to be evaluated.

An interesting aspect of this architecture is that the isolation of the conflicts and the marking of the relations can be often pre-computed without affecting run-time execution costs. Indeed, given a data integration system $\mathcal{I}$, if the pruning module is bypassed, then all the relations are considered relevant as far as the query is concerned, and all the conflicting tuples in the retrieved global database are marked. Obviously, one can also use a finer control on the pruning, e.g., by choosing a set of relations that are relevant for a given workload of queries.

It should be clear that computing the repairs for the whole data integration system is a quite expensive activity. However, as mentioned above it may be done at design-time and after updates (assuming that the latter are infrequent), such that the run-time performance greatly improves. In fact, in order to answer a query, it is then sufficient to rewrite it in terms of suitable SQL statements over the marked database, accessing only those components of a factorization which are relevant for query answering as determined by the relevance analysis. Then, compared to naive evaluation by means of an answer set engine, the execution time is negligible.

7 Experimental Results

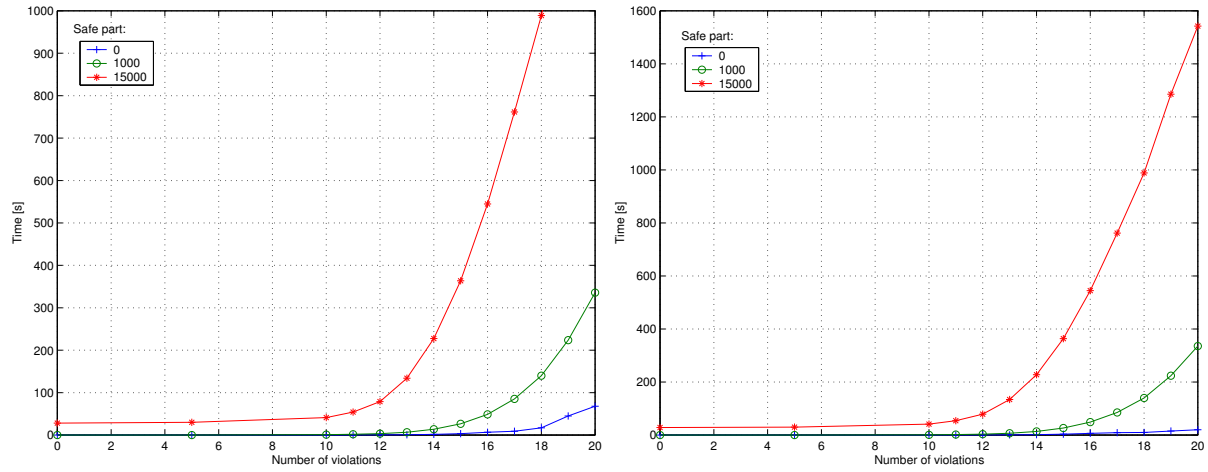
In this section, we present experimental results for evaluating the effectiveness of our approach. The experiments show the benefits of the techniques proposed in the paper (and in particular of exploiting DBMS technology) over implementations of data integration systems completely relying on stable model engines.

7.1 Compared Methods, Benchmark Problems, and Data

In order to evaluate the impact of our technique, we assessed the time needed for query answering when the DLV system computes repairs of the affected part of the retrieved global database only, plus the time required for the recombination of the results in the PostgreSQL relational DBMS, which allows for a convenient encoding of the `ANDBIT`, `INVBIT` and `SUMBIT` operators. This approach to query answering is compared with the standard approach in which the DLV system is used to evaluate a logic encoding of both the data integration system and the query over the whole retrieved global database. Specifically, we report results for the case where the repair encoding is the one discussed in Section 4.2.1; similar performances have been observed by using the other encodings discussed in the paper.

For the comparison, we consider the following benchmark problems:

• In a first set of experiments, we investigate the advantages of our approach, by considering some synthetic data sets and two data integration systems with very simple integrity constraints (namely, one key and one exclusion dependency, respectively) on the global schemas. The results are useful for evidencing



(a) (b)

Figure 7: Stable model computation time in DLV system w.r.t. number of conflicts. (a) One Key. (b) One exclusion dependency.

the potential gain that can be achieved by exploiting "localization" approaches, even in those situations that involve very simple logic programs for computing consistent answers.

- Then, we turn to our running example and present some results on randomly generated data. As a further example, we encoded the classical graph 3-coloring problem, which is well-known to be NP-complete, into querying a data integration system and we show how the performance solutions scales with the size of the graph.
- Finally, we consider a real-life application scenario and show how our techniques may efficiently support data integration in practical applications.

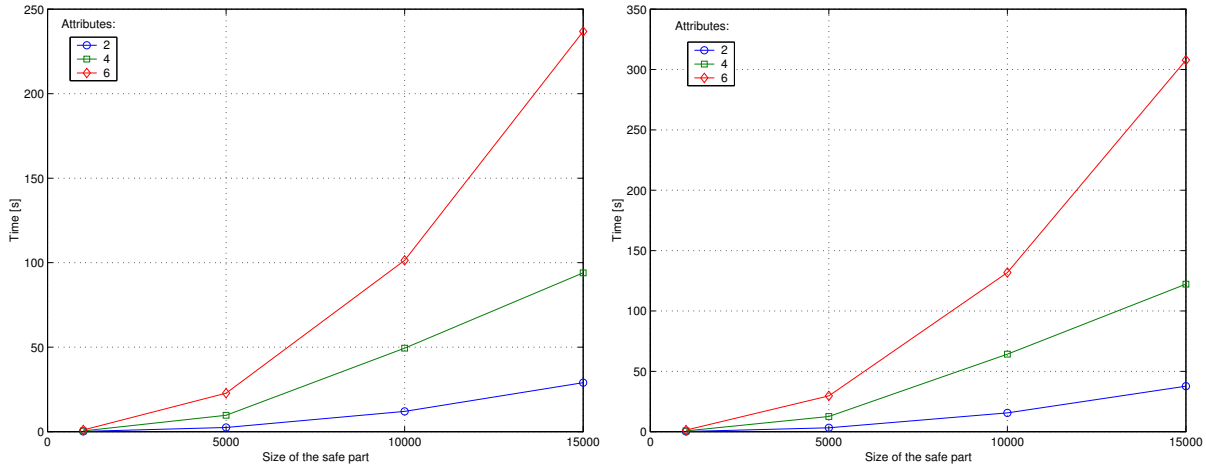
All the experiments have been carried out on a 1.6GHz Pentium IV processor with 512MB memory.

7.2 Testing the Impact of Localization

We built two data integration systems $\mathcal{I}_k$ and $\mathcal{I}_e$, such that the global schema of $\mathcal{I}_k$ contains one predicate p only, having an attribute as a key, and the global schema of $\mathcal{I}_e$ contains two predicates p and q on which an exclusion dependency is issued. Notice that such type of constraints often occur in database design; in particular, exclusion dependencies are typical for database schemes from ER-models and other conceptual data modeling languages, and are widely used in applications in which the global schema is given by an ontology.

Then, we generate some random data in the source databases $\mathcal{D}_k$ and $\mathcal{D}_e$, respectively, by tuning the size of the safe part (i.e., $|\mathcal{S}_{ret(\mathcal{I}_k,\mathcal{D}_k)}|$ and $|\mathcal{S}_{ret(\mathcal{I}_e,\mathcal{D}_e)}|$) and the number of conflicts. In more detail, we assume that each constraint violation involves two facts so that the size of the affected part (i.e., $|\mathcal{A}_{ret(\mathcal{I}_k,\mathcal{D}_k)}|$ and $|\mathcal{A}_{ret(\mathcal{I}_e,\mathcal{D}_e)}|$, respectively) is two times the number of these violations.

In Figure 7, we report the time needed in the DLV system for computing the stable models of the logic program associated to these data integration systems w.r.t. the number of conflicts, for different sizes of the global database (where the sizes of the safe parts are printed). This first set of experiments is particularly relevant, since the cost of computing all the stable models is a reasonable lower-bound for the cost of



(a) (b)

Figure 8: Execution time in DLV system w.r.t. size of the safe part, for different numbers of attributes in relation. (a) One key. (b) One exclusion dependency.

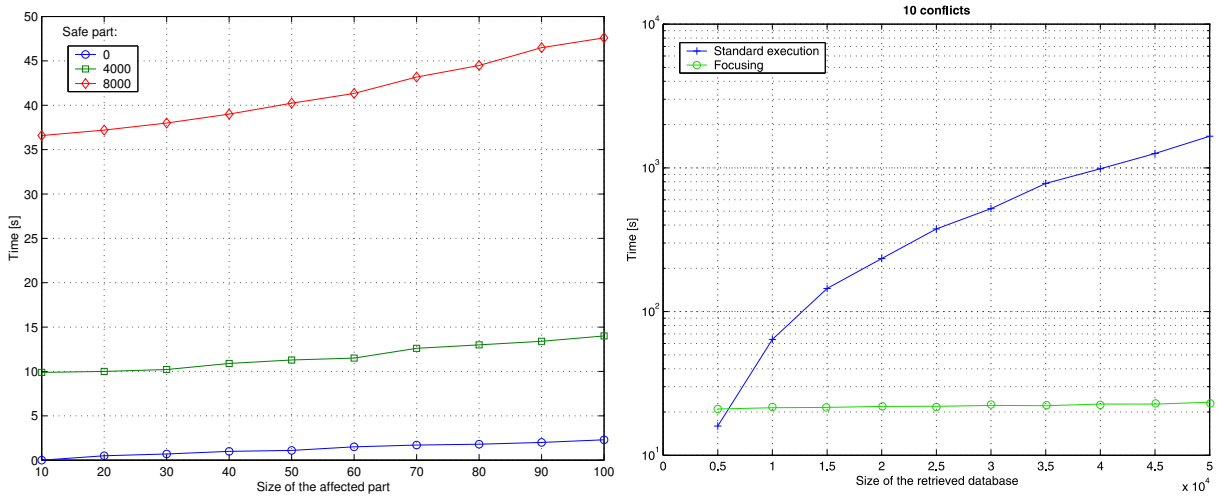
computing the consistent query answers, given that most of the state-of-art answer set engines provide support for "Boolean" query answering, i.e., for deciding whether a given ground fact is entailed in any/all models, but not for computing non-ground queries.

First, we notice that the behavior of DLV is essentially independent from these two types of constraints and scales exponentially in the number of conflicts. Because of this performance degradation, query answering appears to be feasible only for very few conflicts. The main reason for this inefficiency is that the time for computing the stable models strictly depends on the number of the models which we deal with and on their sizes. Then, it suffices to notice that for each conflict added on the global schema we get two different ways for repairing the global database; therefore, given that the size of each repair is about the size of the retrieved global database, the number of processed tuples is generally exponential in the size of the retrieved global database and the approach turns out to be unviable in real scenarios. Note that for a small affected part of the global database (up to log size), in the decomposition approach its repairs can be computed in polynomial time w.r.t. the size of the whole global database (and is generally feasible for constraints from $\mathbf{C}_0$).

The results of Figure 7 stimulated the development of techniques for computing consistent answers even to non-ground queries in stable models engines. Moreover, our preliminary investigations evidenced that answer set engines result quite unpractical for data base applications since they do not offer primitives for interfacing with databases, e.g., for importing and exporting relations or views. And, in fact, in our first experiments it was necessary to write wrappers (e.g., *DB Interface* module) that interface the output of the answer set engines and provides I/O functionalities.

Currently, the DLV system provides instead some interfacing modules to automatically access a relational DBMS by means of standard ODBC calls, and more importantly, provides support for non-ground queries akin to our techniques.

Still, the need for instantiating the logic program for consistent query answering over large data sets makes the use of these systems unfeasible in practice. Indeed, in a second set of experiments, we tested the scaling of DLV in answering non-ground queries. Figure 8 reports the results for evaluating in DLV some non-ground queries on the two scenarios presented above.



(a) (b)

Figure 9: Football Team. (a) Execution time in DLV system w.r.t. size of the affected part. (b) Comparison with the optimization method.

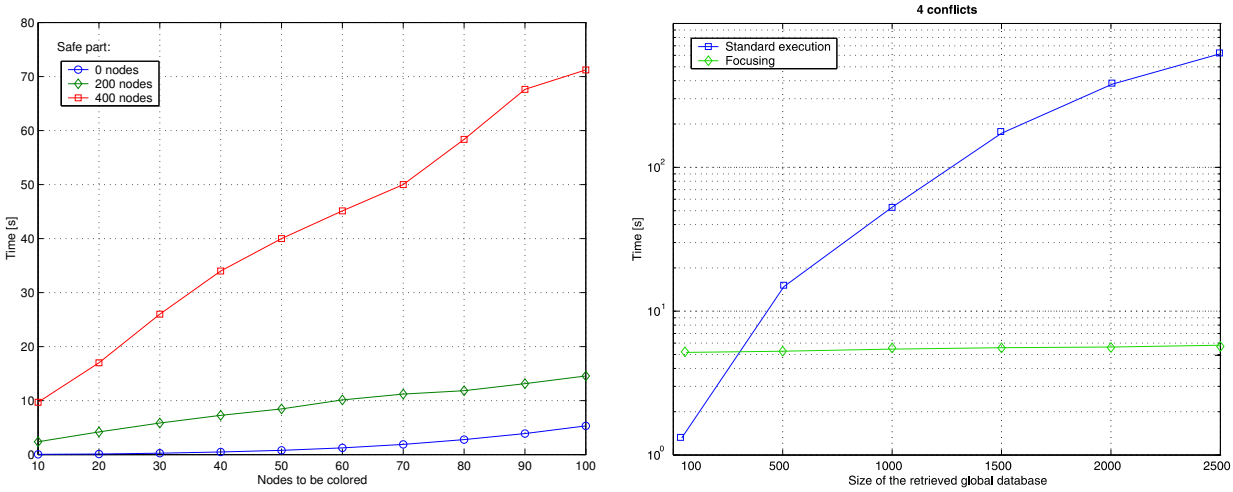
Interestingly, the support for non-ground queries appears to be quite powerful, since the system scales well in the size of the input database for a fixed number of conflicts. However, the performances are not suited for real database applications. In fact, for 15.000 tuples it requires more than 200 seconds for computing answers. Moreover, it is easy to see that the curves rapidly increase if the number of attributes (arities of the relations) grows. This behavior does not correspond to intrinsic complexity of the problem instances, which can be formally proven to be solvable in polynomial time. And, in fact, a careful analysis of the execution time showed that most of the time spent by DLV is for instantiating the logic program for querying the data integration systems.

Our overall approach to optimization of logic programs evaluation tries to face the above problem. Indeed, the localization of the computation can dramatically reduce the size of the program to be instantiated in DLV and, hence, the time needed for the execution.

7.3 Football Teams and 3Coloring

For our running example, we built a synthetic data set $\mathcal{D}_{FT}$, such that tuples retrieved from the sources in $coach$ and $team$ satisfy the key constraints issued on such relation symbols, while retrieved tuples in $player$ violate the corresponding key constraint. 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 $\mathcal{A}_{ret(\mathcal{I}_0,\mathcal{D}_{FT})}$.

For our experiments, we consider the query $Q = \langle q, \mathcal{P}\rangle$ where $\mathcal{P} = \{q(x) \leftarrow player(x, y, z);\ q(x) \leftarrow team(v, w, x)\}$, and we first measure the execution time of the program $\Pi_{\mathcal{I}_0}(q)$, given in Section 4.2.1, in DLV depending on $|\mathcal{A}_{ret(\mathcal{I}_0,\mathcal{D}_{FT})}|$ where $|\mathcal{S}_{ret(\mathcal{I}_0,\mathcal{D}_{FT})}|$ is fixed to the values(i) 0, (ii) 4000, and (iii) 8000, respectively. We stress here that timing for the DLV system refers to query answering over non-ground queries. Thus, the results reported in Figure 9.(a) provide an evidence of the fact that (provided this new functionality) the DLV system scales well w.r.t. the size of the affected part. Still the 'huge' size of the safe part appears to be the most limiting factor for an efficient implementation. Indeed, only 8000 facts (in absence of conflict) would require more than 35 second for consistent query answering.



(a) (b)

Figure 10: 3Coloring. (a) Execution time in DLV w.r.t. number of nodes (i.e., conflicts). (b) Comparison with the optimization method.

The degradation of the performances at the varying of the size of the retrieved global database are further stressed in Figure 9.(b), which shows a comparison (in log-scale) between the consistent query answering by a single DLV program and the optimization approach proposed in this paper. As for the optimization approach, timing includes the cost of computing repairs of the affected database only, plus marking and evaluating the associated SQL query over marked relations. Interestingly, for a fixed number of violations (10 in the figure) the growth of the running time of our optimization method under a varying database size is negligible.

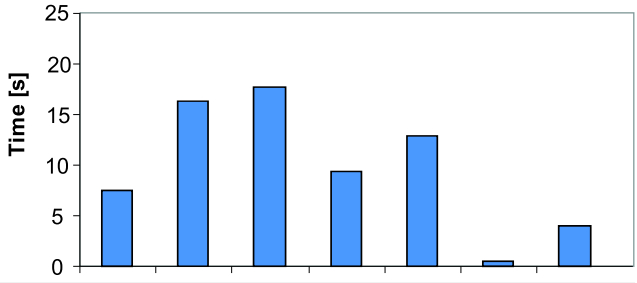
In fact, DLV is able to compute all the models and store them in few milliseconds. This is because each repair consists of about 10 tuples only (this is the size of the affected part), and therefore generating up to 2^{10} repairs of such a size can be efficiently handled by the system. Eventually, computing and storing all the models was assessed to be feasible within 1 second, for scenarios having up to 2^{14} repairs.

Similarly, the time for query evaluation in PostgreSQL itself is negligible. And, in fact, a major share ($\sim$ 20 seconds) is used for marking the repairs and storing them in PostgreSQL, and for rewriting the query. In this respect, our implementation is very prototypical and far from being efficient, because it was just aimed at showing the viability of the focusing approach. We believe that the direct implementation of the marking approach into the architecture of the DLV system (so that DLV can directly compute and store the marked relations) may lead to significant advantages.

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.

As a further example, we encoded the classical NP-complete graph 3-coloring problem into a consistent query answering problem over a data integration system $\mathcal{I}_1 = \langle \mathcal{G}_1, \mathcal{S}_1, \mathcal{M}_1 \rangle$. The global schema $\mathcal{G}_1$ contains the relations $edge(Node, Node)$ and $colored(Node, Color)$, where the attribute $Node$ is established to be the key for $colored$.

For a global database $\mathcal{DB}$, we fixed a number of nodes and generated facts in $edge$ producing a graph; moreover, for each node i, we generated three facts: $colored(i, red)$, $colored(i, blue)$, $colored(i, yellow)$.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
N° global relations	2	2	1	1	2	1	1
N° source relations	8	5	5	2	5	2	5
Size	161250	208828	143282	270047	381001	817	143282

Figure 11: Timing of query evaluation in the INFOMIX Demo Scenario.

Clearly $\mathcal{DB}$ is inconsistent with the key constraint on the relation $colored$, and each node creates three conflicts. It is easy to see that only one of the three facts involved in each constraint violation can be maintained in each repair of $\mathcal{DB}$.

Now, consider the query $q \leftarrow edge(x, y), colored(x, C), colored(y, C)$. As easily seen, it evaluates to true on $\mathcal{DB}$ iff there is no legal 3-coloring for the graph in $\mathcal{DB}$.

Figure 10.(a) reports the execution time in DLV for different values of the size of the affected part, while Figure 10.(b) reports the comparison with our approach. Again, the advantage of the localization technique is evident when the size of the database increases.

7.4 Demo Scenario

We finally present some experiments which we have carried out on the demonstration scenario of the INFOMIX project on advanced information integration, supported by the IST programme of European Commission (IST-2001-33570). This scenario refers to the information system of the University "La Sapienza" in Rome. The global schema consists of 14 global relations with 29 integrity constraints, while the source schema includes 29 relations (in 3 legacy databases) and 12 web wrappers (generating relational data) for more than 24MB of data regarding students, professors and exams in any faculty of the University. On this schema, some typical queries with peculiar characteristics are formulated, which model different use cases. A detailed description of the scenario is given in [41].

In Figure 11, for each query Q the number of global relations involved in Q, the number of sources required to be accessed for answering Q, and the size of the retrieved global database which is relevant for Q are shown. The evaluation times reported are those under our optimization approach.

It can be easily seen that the database sizes are far larger than the ones used for some preliminary tests. Consequently, naive query evaluation in DLV (even with the support for non-ground queries) would take three orders of magnitude longer, and is thus far from feasibility. However, in this application we experienced that rather few constraint violations (up to a dozen) occurred in the part relevant for query answering (especially, after data cleaning is performed; clearly, the number of constraint violations in the whole retrieved global database is much higher). Handling the associated affected database in DLV and processing the rewritten query over the DBMS system is feasible in few seconds.

8 Discussion and Conclusion

In this paper, we have presented methods and techniques to optimize query answering in advanced data integration systems which deal with possibly inconsistent or incomplete data, based on non-monotonic logic programming. To this end, we set up a formal framework for data integration and query answering which accommodated several proposals for repairs semantics, including [32, 2, 4, 8, 14, 16, 36, 19, 53, 52, 34], and have considered at a generic level logic-programming specifications for it which are hierarchically composed of different modules. We then have developed optimization methods and techniques of three kinds: methods genuine to logic programming (detection of rules relevant to query answering), methods for the problem in the framework as such (localization and decomposition), and a technique for combining a logic programming and a relational database engine. In fact, the latter combination technique is not limited to logic programming engines and might be exploited for other hybrid system approaches as well.

We point out here that our methods and results can be extended in different directions.

• Firstly, the localization method and results can be extended to repair semantics based on preference orderings which do not satisfy the properties ($\star$), (†), and (‡) in Section 3.2 as well. For example, Chomicki and Marcinkowski [18] consider repairs in which a smallest (in terms of inclusion) set of tuples is deleted from the database but no tuples are added. For such repairs, Proposition 5.2, Lemma 5.3, and the main result on localization (Theorem 5.4) can be established similarly.

Furthermore, answering a positive query $Q = \langle q, \mathcal{P} \rangle$ (i.e., the program $\mathcal{P}$ does not contain negation) on all repairs with respect to some preorder is equivalent to answering it only on those repairs $\mathcal{R}$ which are minimal under set-inclusion, i.e., do not contain any other repair $\mathcal{R}'$ properly. Therefore, even if a preorder $\leq_{\mathcal{DB}}$ fails to satisfy ($\star$), (†), and (‡), it may be possible to characterize the set-inclusion minimal repairs w.r.t. $\leq_{\mathcal{DB}}$ as repairs under a different ordering $\leq'_{\mathcal{DB}}$ which satisfies these properties. A particular example is the preorder of loosely sound semantics [16, 17], which is given by $\mathcal{R}_1 \sqsubseteq_{\mathcal{DB}} \mathcal{R}_2$ if and only if $\mathcal{R}_1 \cap \mathcal{DB} \supseteq \mathcal{R}_2 \cap \mathcal{DB}$. This orderding clearly violates Property ($\star$). We can use here the related preorder $\mathcal{R}_1 \sqsubseteq'_{\mathcal{DB}} \mathcal{R}_2$ defined by $\mathcal{R}_1 \sqsubseteq_{\mathcal{DB}} \mathcal{R}_2 \wedge (\mathcal{R}_1 \cap \mathcal{DB} = \mathcal{R}_2 \cap \mathcal{DB} \Rightarrow \mathcal{R}_1 \backslash \mathcal{DB} \subseteq \mathcal{R}_2 \backslash \mathcal{DB})$, for instance.

• Secondly, the method and results can also be extended to certain data integration systems which do not match the basic framework in this paper. For example, [45, 17] address the repair problem in GAV systems with existentially quantified inclusion dependencies and key constraints on the global schema. In these papers, techniques for suitably reformulating user queries in order to eliminate inclusion dependencies are presented, which leads to a rewriting that can be evaluated on the global schema taking only key constraints into account. We point out that, provided such a reformulation, the logic specifications proposed in [45, 17] fit our framework.

• Furthermore, the logic formalization of LAV systems proposed in [12, 14] 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, like in our GAV framework, the repairs are computed in [12, 14] from the stable models of a suitable logic program, as well as in [15].

• Finally, notice that our marking approach makes it possible to materialize a kind of condensed representation for the repairs of a data integration system (as discussed in Section 6.4), over which consistent query answering is possible by means of a suitable query rewriting process. This is particularly useful when the focus is on data materialization rather than on query answering, as it happens in the context of Data Exchange (cf. [30, 31]). In Data Exchange, the notion of the *core* of a relational structure has been proved to

be the most appropriate choice for materialization, and mechanisms to compute the core have been provided for the case when constraints over the target schema consist of tuple-generating and equality-generating dependencies and when source-to-target dependencies are formalized in the GLAV approach, a generalization of both LAV and GAV, [31]. In this respect, the marked database described in the present paper represents a viable way to materialization when universal constraints are issued over the target schema, and when source-to-target dependencies are formalized in the GAV approach.

At present, to our knowledge there are only prototype implementations of data integration (or database) systems which fit the semantic repair framework available. The most noticeable among them are the INFOMIX system [47], Hippo [20, 19], and ConQuer [33].

The INFOMIX system [47] has been developed as a proof of concept prototype for an advanced data integration system based on computational logic technology, and in particular on non-monotonic logic programming. The technical results of this paper have been driven by the research for optimization methods to make such an approach computationally feasible, and found their way in different form into the prototype. Noticeably, the INFOMIX system is not the implementation of all results in this paper.Indeed, the system exploits the Relevance Pruning step for pruning the rules that are not relevant for computing answers, but then it consider a novel variant of magic sets tailored for data integration [27] rather than our localization approach. Importantly, the magic sets technique is orthogonal to the methods presented there and can be profitably coupled with them. Indeed, magic sets are mainly aimed at reducing the set of facts on which the engine has to work by eventually "pushing" constants from the query into the rules, but they do not distinguish between safe and affected facts (thus, if no constant can be pushed, then magic sets will work on the whole set of retrieved data). An interleaved usage of both techniques deserves further investigation.

The Hippo system [19, 20] has been designed for answering SJUD queries (i.e., definable in relational algebra without projection) under denial constraints from a possibly inconsistent relation, using a repair semantics which deletes a smallest (under set inclusion) set of tuples to restore consistency (i.e., no tuples are added). The repairs are computed by the use of algorithms on a conflict hypergraph which is built from constraint violations. Query answering in Hippo is guaranteed in polynomial time (measured in data complexity). The system incorporates techniques for sound and complete approximations of the query result to reduce the number of candidate tuples for which membership in the result is checked, as well as techniques for reducing the effort in testing such membership.

ConQuer [33] is a system for computing consistent answers from large inconsistent databases, under a repair semantics relying on symmetric set difference-based ordering. In the line of a previous work of some of the ConQuer authors [34], the focus in this system is on tractable cases of the consistent query answering problem, which can be solved by suitably reformulating user queries into SQL, whereas solutions for intractable (more general) instances of the problem are not provided. In particular, ConQuer considers database schemas on which only key dependencies are allowed, and user queries which belong to a subset of the class of conjunctive queries, possibly enriched with aggregates. ConQuer is showed to be very efficient in providing consistent answers over large databases with many inconsistent tuples (some of the experiments in [33] involve a database with an affected portion ranging up to the 50 per cent of the whole database). The technique presented in [34] has been recently extended in [39] to deal with also exclusion dependencies issued on a database schema.

We point out that the optimization strategies developed here are orthogonal to those provided in the Hippo and the ConQuer systems, which are geared towards highly efficient query answering for specific, polynomial-time classes of queries. Our results, instead, aim at supporting more general, highly expressive classes of queries, including also queries intractable under worst case complexity. For instance, out of the 7

queries of the INFOMIX demo scenario described in Section 7.4, only one can be expressed in Hippo, while ConQuer is not applicable for the specific (complex) constraint setting.

Several issues remain for further work. Our experiments and related ones on the INFOMIX prototype provide some evidence for the effectiveness of the techniques presented in this paper. Given that the classes of queries expressions that can be handled by Hippo and ConQuer can be recognized efficiently, the results of [19, 34] and ours may be fruitfully combined in a system which facilitates scalable data integration.

Another issue concerns an extension of our techniques to more expressive classes of queries. To enable benefits from the usage of a relational database engine in query answering, the query class must be handled by the relational DBMS; therefore, recursive queries with stratified negation (at most) can be accomodated in this way, under further restrictions dependent on the support of the SQL99 standard by a DBMS.

Furthermore, benchmarking of inconsistent data integration (cf. also [19]) needs more attention. While currently, integration scenarios –both synthetic and realistic ones– are created by different researchers on an ad hoc basis, it would be useful to have an acknowledged repository of benchmark scenarios for testing and evaluating implemented data integration systems. In particular, meaningful generation of inconsistencies is an interesting issue here.

Finally, it would be interesting to see to what extent a framework similar to ours can be established for more general data integration systems, such as GLAV systems.

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 SIGACT SIGMOD SIGART Symp. 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. Conf. on Flexible Query Answering Systems (FQAS 2000)*, pp. 27–41. Springer, 2000.

[4] M. Arenas, L. E. Bertossi, and J. Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming*, 3(4):393–424, 2003. arXiv.org paper cs.DB/0207094.

[5] O. Arieli, M. Denecker, B. V. Nuffelen, and M. Bruynooghe. Coherent integration of databases by abductive logic programming. *Journal of Artificial Intelligence Research*, 21:245–286, 2004.

[6] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. Int'l Symposium on Principles of Database Systems (PODS 1986)*, pp. 1–16, 1986.

[7] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Univ. Press, 2002.

[8] P. Barceló and L. E. Bertossi. Logic programs for querying inconsistent databases. In *Proc. Symposium on Practical Aspects of Declarative Languages (PADL'03)*, pp. 208–222, 2003.

[9] C. Beeri and R. Ramakrisnhan. On the power of magic. *Journal of Logic Programming*, 10(1–4):255–259, 1991.

[10] A. Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 102–110. ACM Press, 2003.

[11] L. Bertossi and J. Chomicki. Query answering in inconsistent databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*, chapter 2, pp. 43–83. Springer, 2003.

[12] L. E. Bertossi, J. Chomicki, A. Cortes, and C. Gutierrez. Consistent answers from integrated data sources. In *Proc. 6th Int. Conf. on Flexible Query Answering Systems (FQAS 2002)*, pp. 71–85, 2002.

[13] M. Bouzeghoub and M. Lenzerini. Introduction to the special issue on data extraction, cleaning, and reconciliation. *Information Systems*, 26(8):535–536, 2001.

[14] L. Bravo and L. Bertossi. Logic programming for consistently querying data integration systems. In *Proc. 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pp. 10–15, 2003.

[15] L. Bravo and L. Bertossi. Disjunctive deductive databases for computing certain and consistent answers to queries from mediated data integration systems. *Journal of Applied Logic*, 3(1):329-367, 2005.

[16] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proc. 22nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2003)*, pp. 260–271, 2003.

[17] A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, pp. 16–21, 2003.

[18] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197((1-2)):90–121, 2005.

[19] J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. In *Proc. 13th ACM Conference on Information and Knowledge Management (CIKM-2004)*, pp. 417–426. ACM Press, 2004.

[20] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A system for computing consistent answers to a class of sql queries. In *9th International Conference on Extending Database Technology (EDBT-2004)*, pp. 841–844. Springer Verlag, 2004.

[21] C. Cumbo, W. Faber, G. Greco, and N. Leone. Enhancing the magic-set method for disjunctive datalog programs. In *Proc. 20th International Conference on Logic Programming – ICLP'04*, pp. 371–385, Saint-Malo, France, Sept. 2004.

[22] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[23] J. Dix. Semantics of logic programs: Their intuitions and formal properties. An Overview. In *Logic, Action and Information. Proc. Konstanz Colloquium in Logic and Information (LogIn'92)*, pp. 241–329. DeGruyter, 1995.

[24] J. Dzifčák. Optimization of INFOMIX repair programs under Answer-Set Programming Paradigm. Master's thesis, Dept. Mathematics, Physics and Informatics, Comenius University Bratislava, Slovakia, 2005.

[25] T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data integration systems. In C. Palamidessi, editor, *Proc. 19th International Conference on Logic Programming (ICLP 2003)*, number 2916 in LNCS, pp. 163–177. Springer, 2003.

[26] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[27] W. Faber, G. Greco, and N. Leone. Magic sets and their application to data integration. In T. Eiter and L. Libkin, editors, *Proc. 10th International Conference on Database Theory (ICDT 2005)*, number 3363 in LNCS, pp. 306–320. Springer, 2005.

[28] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In *Proc. 17th International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pp. 635–640, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

[29] W. Faber, N. Leone, and G. Pfeifer. Optimizing the computation of heuristics for answer set programming systems. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proc. 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, Vienna, Austria, September 2001, number 2173 in Lecture Notes in AI (LNAI), pp. 288–301. Springer Verlag, 2001.

[30] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005. Special issue for selected papers from the 2003 International Conference on Database Theory.

[31] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. In *PODS '03: Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 90–101, New York, NY, USA, 2003. ACM Press.

[32] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *Proc. 2nd ACM SIGACT SIGMOD Symp. on Principles of Database Systems (PODS'83)*, pp. 352–365, 1983.

[33] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD Conference*, 2005.

[34] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. In T. Eiter and L. Libkin, editors, *Proc. 10th International Conference on Database Theory (ICDT 2005)*, number 3363 in LNCS, pp. 337–351. Springer, 2005.

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

[36] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. on Knowledge and Data Engineering*, 15(6):1389–1408, 2003.

[37] S. Greco. Binding propagation techniques for the optimization of bound disjunctive queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):368–385, March/April 2003.

[38] S. Greco, D. Sacca, and C. Zaniolo. The pushdown method to optimize chain logic programs (extended abstract). In *Proc. Int'l Colloquim on Automata, Languages and Programming (ICALP)*, pp. 523–534, 1995.

[39] L. Grieco, D. Lembo, R. Rosati, and M. Ruzzi. Consistent query answering under key and exclusion dependencies: algorithms and experiments, 2005. submitted for publication to an international conference.

[40] A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in datalog extensions. *Journal of the ACM*, 48(5):971–1012, 2001.

[41] INFOMIX Project Team. Demo Scenario. Technical Report INFOMIX S7-1, INFOMIX Project Consortium, June 2004. Available at `http://sv.mat.unical.it/infomix`.

[42] D. B. Kemp, D. Srivastava, and P. J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, July 1995.

[43] M. Kifer and E. L. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, 1992.

[44] R. A. Kowalski and F. Dadri. Logic programming with exceptions. In *Proc. 7th Int. Conf. on Logic Programming (ICLP'90)*, pp. 490–504, 1990.

[45] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *Proc. 9th Int. Workshop on Knowledge Representation meets Databases (KRDB 2002)*. `http://ceur-ws.org/Vol-54/`, 2002.

[46] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2002)*, pp. 233–246, 2002.

[47] N. Leone, T. Eiter, W. Faber, M. Fink, G. Gottlob, G. Greco, G. Ianni, E. Kalka, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, R. Rosati, M. Ruzzi, W. Staniszkis, and G. Terracina. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proc. ACM SIGMOD/PODS 2005 Conference, June 13-16,2005, Baltimore, Maryland*. pp. 915–917, 2005. Demo paper.

[48] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 2004. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`. DLV Homepage: `http://www.dbai.tuwien.ac.at/proj/dlv`.

[49] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[50] V. Lifschitz and H. Turner. Splitting a logic program. In P. Van Hentenryck, editor, *Proc. 11th International Conference on Logic Programming (ICLP'94)*, pp. 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[51] F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In D. Fensel, F. Giunchiglia, D. McGuiness, and M.-A. Williams, editors, *Proc. 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02)*, pp. 170–176. Morgan Kaufmann, 2002.

[52] J. Lin. Integration of weighted knowledge bases. *Artificial Intelligence*, 83(2):363–378, 1996.

[53] J. Lin and A. O. Mendelzon. Merging databases under constraints. *Int. J. of Cooperative Information Systems*, 7(1):55–76, 1998.

[54] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[55] J. Minker and D. Seipel. Disjunctive logic programming: A survey and assessment. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future*, number 2407 in LNCS/LNAI, pp. 472–511. Springer Verlag, 2002. Festschrift in honour of Bob Kowalski.

[56] B. V. Nuffelen, A. Cortés-Calabuig, M. Denecker, O. Arieli, and M. Bruynooghe. Data integration using id-logic. In A. Persson and J. Stirna, editors, *Proc. 16th International Conference on Advanced Information Systems Engineering (CAiSE 2004)*, LNCS 3084, pp. 67–81. Springer, 2004.

[57] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and M. Y. Vardi. Logical query optimization by proof-tree transformation. *Journal of Computer and System Sciences*, 47(1):222–248, 1993.

[58] K. A. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[59] Y. Sagiv. Optimizing datalog programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 659–698. Morgan Kaufmann, 1988.

[60] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In H. V. Jagadish and I. S. Mumick, editors, *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pp. 435–446. ACM Press, June 1996.

[61] O. Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15(3):231–242, 1993.

[62] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, June 2002. Smodels home page: `http://www.tcs.hut.fi/Software/smodels/`.

[63] P. J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proc. 13th Symposium on Principles of Database Systems (PODS'94)*, pp. 56–67. ACM Press, May 1994.

[64] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.