Repair Localization for Query Answering from Inconsistent Databases

THOMAS EITER and MICHAEL FINK
Technische Universität Wien
GIANLUIGI GRECO
Università della Calabria
and
DOMENICO LEMBO
SAPIENZA Università di Roma

Query answering from inconsistent databases amounts to finding "meaningful" answers to queries posed over database instances that do not satisfy integrity constraints specified over their schema. A declarative approach to this problem relies on the notion of repair, that is, a database that satisfies integrity constraints and is obtained from the original inconsistent database by "minimally" adding and/or deleting tuples. Consistent answers to a user query are those answers that are in the evaluation of the query over each repair. Motivated by the fact that computing consistent answers from inconsistent databases is in general intractable, the present paper investigates techniques that allow to localize the difficult part of the computation on a small fragment of the database at hand, called "affected" part. Based on a number of localization results, an approach to query answering from inconsistent data is presented, in which the query is evaluated over each of the repairs of the affected part only, augmented with the part that is not affected. Single query results are then suitably recombined. For some relevant settings, techniques are also discussed to factorize repairs into components that can be processed independently of one another, thereby guaranteeing exponential gain w.r.t. the basic approach, which is not based on localization. The effectiveness of the results is demonstrated for consistent query answering over expressive schemas, based on logic programming specifications as proposed in the literature.

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

Some results in this paper appeared, in preliminary form, in Eiter et al. [2003].

Authors' addresses: T. Eiter, M. Fink, Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria; email: eiter@kr.tuwien.ac.at; michael@kr.tuwien.ac.at; G. Greco, Dipartimanto di Matematica, Università della Calabria, Via Pietro Bucci 30B, I-87036 Rende, Italy; email: ggreco@mat.unical.it; D. Lembo, Dipartimento di Informatica e Sistemistica, SAPIENZA Università di Roma, Via Ariosto 25, I-00185 Roma, Italy; email: lembo@dis.uniromal.it.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing, relational databases*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Answer/reason extraction, inference engines, logic programming, nonmonotonic reasoning and belief revision*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational logic*

General Terms: Experimentation, Languages, Theory

Additional Key Words and Phrases: Database repairs, inconsistency management in databases, consistent query answering, stable models, logic programming, data integration

ACM Reference Format:

Eiter, T., Fink, M., Greco, G., Lembo, D. 2008. Repair localization for query answering from inconsistent databases. ACM Trans. Datab. Syst. 33, 2, Article 10 (June 2008), 51 pages. DOI = 10.1145/1366102.1366107 http://doi.acm.org/ 10.1145/1366102.1366107

1. INTRODUCTION

A database is inconsistent if it does not satisfy the integrity constraints specified over its schema. This may happen for different reasons [Arenas et al. 2003]; for instance, when preexisting data are reorganized under a new schema that has integrity constraints describing semantic aspects of the new scenario. This is particularly challenging in the context of data integration, where a number of data sources, heterogeneous and widely distributed, must be presented to the user as if they were a single (virtual) centralized database, which is often equipped with a rich set of constraints expressing important semantic properties of the application at hand. Since, in general, the integrated sources are autonomous, the data resulting from the integration are likely to violate these constraints.

One of the main issues arising when dealing with inconsistent databases is establishing the answers which have to be returned to a query issued over the database schema.

Example 1.1. Consider a database schema χ_0 providing information about soccer teams of the 2006/07 edition of the U.E.F.A. Champions League. The schema 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.

Consider the following inconsistent database D_0 for χ_0 (possibly built by integrating some autonomous data sources):

$player^{D_0}$:

10	*Totti*	*RM*
9	*Ronaldinho*	*BC*

$team^{D_0}$:

RM	*Roma*	10
BC	*Barcelona*	8
RM	*Real Madrid*	10

$coach^{D_0}$:

7	*Capello*	*RM*

D_0 violates the key constraint on *team*, witnessed by the facts $team(RM, Roma, 10)$ and $team(RM, Real\ Madrid, 10)$, which coincide on *Tcode* but differ on *Tname*. In such a situation, it is not clear what answers should be returned to a query over D_0 asking, for instance, for the names of teams, or for the pairs formed by team code and team leader.

$player^{R_1}$:

10	*Totti*	*RM*
9	*Ronaldinho*	*BC*

$team^{R_1}$:

RM	*Roma*	10
BC	*Barcelona*	8

$coach^{R_1}$:

7	*Capello*	*RM*

$player^{R_2}$:

10	*Totti*	*RM*
9	*Ronaldinho*	*BC*

$team^{R_2}$:

BC	*Barcelona*	8
RM	*Real Madrid*	10

$coach^{R_2}$:

7	*Capello*	*RM*

Fig. 1. Repairs of D_0.

The standard approach to remedy the existence of conflicts in the data is through data cleaning [Bouzeghoub and Lenzerini 2001]. This approach is procedural in nature, and is based on domain-specific transformation mechanisms applied to the data. One of its problems is incomplete information on how certain conflicts should be resolved [Staworko et al. 2006]. This typically happens in systems which are not tailored for business logic support at the enterprise level, like systems for information integration on-demand. Here, data cleaning may be insufficient even if only few inconsistencies are present in the data.

In the last years, an alternative declarative approach has been investigated which builds on the notion of a *repair* for an inconsistent database [Arenas et al. 1999]. Roughly speaking, a repair is a new database which satisfies the constraints in the schema and minimally differs from the original one. The suitability of a possible repair depends on the underlying semantics adopted for the inconsistent database, and on the kinds of integrity constraints which are allowed on the schema. Importantly, in general, not a single but multiple repairs might be possible; therefore, the standard way of answering a user query is to compute the answers that are true in every possible repair, called *consistent answers* in the literature.

Example 1.2. Recall that in our scenario, the database D_0 for χ_0 violates the key constraint on *team*, witnessed by *team*(*RM*, *Roma*, 10) and *team*(*RM*, *Real Madrid*, 10).

According to Arenas et al. [1999], a repair results by removing exactly one of these facts from D_0. Hence, there are two repairs only, say R_1 and R_2, which are as shown in Figure 1. Accordingly, the consistent answer to the query asking for the names of the teams is {(*Barcelona*)}, while the consistent answers to the query asking for pairs of team code and team leader are {(*RM*, *10*), (*BC*, *8*)}.

Query answering in the presence of inconsistent data (aka consistent query answering) has been the subject of a large body of research (for a survey on this topic, see Bertossi and Chomicki [2003], and for a discussion on relevant issues in the area see Chomicki [2007]), and some prototype implementations of systems which fit the semantic repair framework are available [Fuxman et al. 2005; Chomicki et al. 2004b; Leone et al. 2005]. Basically, these systems differ in the kinds of constraints and queries they are able to deal with. Indeed, depending on these two ingredients, the data complexity of consistent query answering ranges from poynomial time over co-NP up to Π_2^P [Calì et al. 2003a; Chomicki and Marcinkowski 2005; Chomicki 2007].

1.1 Contributions

In this article, we elaborate techniques for consistent query answering in highly expressive settings. Given that in these cases query answering is unlikely to be feasible in polynomial time, our main research interest is to devise an approach that allows to localize the "difficult" part of the computation in a small fragment of the database at hand.

The basic intuition of this approach is that resolving constraint violations in inconsistent databases does not generally require to deal with the whole set of facts. For instance, in Example 1.1 inconsistency may be fixed by just looking at the (few) tuples conflicting on the key. However, there are many interesting cases for which devising some similar strategies is not as simple as that and, therefore, it appears relevant to assess under which circumstances a localization approach can be pursued and when localized repair computation can be exploited to optimize consistent query answering. In this respect, our overall contribution is twofold in nature.

First, we attack the problem from a theoretic point of view. We provide a unifying view of previous approaches to query answering from inconsistent data, we shed light on the interaction between integrity constraint violation and the structure of repairs, and we study localization and factorization of consistent query answering. Specifically,

1) We present a formal framework for consistent query answering which is, to large extent, independent of a commitment to a specific definition of repair, but is based on a common setting of repair semantics: the repairs of the database are characterized by the minimal (nonpreferred) databases from a space of candidate repairs with a preference order. Our setting generalizes previous proposals in the literature, such as set-inclusion based orderings [Fagin et al. 1983; Arenas et al. 1999, 2003; Barceló and Bertossi 2003; Bravo and Bertossi 2003; Calì et al. 2003a, 2003b; Chomicki and Marcinkowski 2005; Greco et al. 2003], cardinality-based orderings [Arenas et al. 2003; Lin and Mendelzon 1998], and weighted-based orderings [Lin 1996].
2) We investigate some locality properties for repairing inconsistent databases, aiming to isolate in the data those facts that will possibly be touched by a repair, called the "affected part" of the database and the facts that for sure will be not, called the "safe part" of the database. Specifically, we establish localization results for different classes of constraints:
   —The first class, $\mathbf{C}_0$, 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 over database relations and $\phi(\vec{x})$ is a disjunction of built-in literals. These constraints are semantically equivalent to denial constraints [Chomicki et al. 2004a].
   —The second class, $\mathbf{C}_1$, allows more general constraints of the 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 over database relations.
   —The third class, $\mathbf{C}_2$, has similar constraints $\forall\vec{x}\alpha(\vec{x}) \supset \beta(\vec{x}) \vee \phi(\vec{x})$; here $\alpha(\vec{x})$ may be empty, representing unconditional logical truth, but $\beta(\vec{x})$ may have at most one atom.
   —The fourth class is the class of all universal constraints in clausal form. Thus, semantically, this class captures all universal constraints.

3) We propose a *repair localization approach* to query answering from inconsistent databases, in which the query is first evaluated over each of the repairs of the affected part only, augmented with the safe part, and then results are suitably recombined. Also, we investigate techniques for factorizing repairs into components that can be processed independently of each other. For some relevant settings, these techniques guarantee an exponential gain compared to the basic approach. Indeed, by looking at the actual forms of inconsistencies occurring in the underlying database instance, we identify settings for which the number of localized repairs to be considered for consistent query answering is linear in the size of the affected part, while generic database instances have data complexity from co-NP to Π_2^P [Calì et al. 2003a; Chomicki and Marcinkowski 2005; Chomicki 2007; Greco et al. 2003].

   Secondly, our contribution is practical. Indeed, based on the preceding localization results, we develop strategies to consistent query answering relying on existing technologies offered by stable model engines and relational DBMSs. Resembling several proposals in the literature, our techniques make use of logic programs to solve inconsistency. However, we limit their usage to the affected part of the data. This approach is useful to localize the difficult part of the computation and to overcome the lack of scalability of current (yet still improving) implementations of stable model engines such as DLV [Leone et al. 2006] or Smodels [Simons et al. 2002]. Specifically:

4) We propose a formal model of inconsistency resolution via logic programming specification, which abstracts from several proposals in the literature [Arenas et al. 2003; Barceló and Bertossi 2003; Bertossi et al. 2002; Bravo and Bertossi 2003; Calì et al. 2003b; Greco et al. 2003]. Results obtained on this model are applicable to all such approaches.
5) We discuss an architecture that recombines the repairs of the affected part with the safe part of an inconsistent database, interleaving a stable model 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 help to achieve scalability. In this architecture, the database engine has to "update" the consistent answers to a certain query each time a new repair is computed by the stable model engine. To further improve this strategy, a technique for simultaneously processing a (large) group of repairs in the DBMS is proposed. Basically, it consists in a marking and query rewriting strategy for compiling the reasoning tasks needed for consistent query answering into a relational database engine.
6) Finally, we assess the effectiveness of our approach in a suite of experiments. They have been carried out on a prototype implementation in which the stable model engine DLV is coupled with the DBMS PostgreSQL. The experimental results show that the implementation scales reasonably well.

We observe that our results on localization extend and generalize previous localization results that have been utilized (sometimes tacitly) for particular repair orderings and classes of constraints, for instance, for denial constraints and repairs that are closest to the original database measured by set symmetric difference [Chomicki et al. 2004a]. Also, our results can be exploited for efficient implementation of consistent query answering techniques in general, independent of a logic-programming based approach.

The rest of this article is organized as follows. Section 2 introduces the notation for the relational data model and for logic programs used throughout the paper. Section 3 defines the formal framework for consistent query answering from inconsistent databases. Localization properties in database repairs and their exploitation to optimize consistent query answering are discussed in Section 4 and Section 5, respectively. The logic specification for consistent query answering is presented in Section 6, together with an architecture that interleaves a DBMS and a stable model engine. Section 7 considers other approaches to consistent query answering, and Section 8 reports results of our experimental activity. The final Section 9 concludes the paper.

Some further details of our techniques have been moved to an on-line appendix, which also contains further examples and experiments.

2. PRELIMINARIES

2.1 Data Model

We assume a countable infinite database domain $\mathcal{U}$ whose elements are referenced by constants $c_1, c_2, \ldots$ under the *unique name assumption*, that is, different constants denote different real-world objects.

A *relational schema* (or simply *schema*) χ 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. We consider here universally quantified constraints [Abiteboul et al. 1995], that is, first-order sentences of the form

$$\forall \vec{x}\, A_1(\vec{x}_1) \wedge \cdots \wedge A_l(\vec{x}_l) \supset B_1(\vec{y}_1) \vee \cdots \vee B_m(\vec{y}_m) \vee \phi_1(\vec{z}_1) \vee \cdots \vee \phi_n(\vec{z}_n), \quad (1)$$

where $l + m > 0$, $n \geq 0$, the $A_i(\vec{x}_i)$ and the $B_j(\vec{y}_j)$ are atoms over Ψ, the $\phi_k(\vec{z}_k)$ are atoms or negated atoms over possible built-in relations like equality ($=$), inequality ($\neq$), etc., $\vec{x}$ 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 from $\vec{x}$ and constants from $\mathcal{U}$.[1] The conjunction left of "$\supset$" is the *body* of the constraint, and the disjunction right of "$\supset$" its *head*.

In the rest of the article, $\chi = \langle \Psi, \Sigma \rangle$ denotes a relational schema. Since all variables in (1) are universally quantified, we omit quantifiers in constraints.

Note that (1) is a clausal normal form for arbitrary universal constraints on a relational schema. However, since existential quantification is not allowed, referential constraints such as *foreign-key constraints* cannot be expressed. We

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

pay special attention to the following subclasses of constraints:

—Constraints with only built-in relations in the head (i.e., $m = 0$ in (1)). The class of these constraints, which we denote by $\mathbf{C}_0$, is a clausal normal form of *denial constraints* [Chomicki et al. 2004a], also called *generic constraints* in Bertossi and Chomicki [2003]. This class (semantically) includes:
—*key constraints* $p(\vec{x}, \vec{y}) \wedge p(\vec{x}, \vec{z}) \supset y_i = z_i$, for $1 \leq i \leq n$,
—*functional dependencies* $p(\vec{x}, \vec{y}, \vec{v}) \wedge p(\vec{x}, \vec{z}, \vec{w}) \supset y_i = z_i$, for $1 \leq i \leq n$, and
—*exclusion dependencies* $p_1(\vec{v}, \vec{y}) \wedge p_2(\vec{w}, \vec{z}) \supset y_1 \neq z_1 \vee \cdots \vee y_n \neq z_n$,
where $\vec{y} = y_1, \ldots, y_n$ and $\vec{z} = z_1, \ldots, z_n$.

—Constraints with nonempty body (i.e., $l > 0$ in (1)). 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 $l + m > 0$). The class $\mathbf{C}_1$ includes, for instance, *inclusion dependencies* of the form $p_1(\vec{x}) \supset p_2(\vec{x})$.

—Constraints with at most one database atom in the head (i.e., $m \leq 1$ in (1)). We denote the class of these constraints, which we call nondisjunctive, by $\mathbf{C}_2$. Beyond denials, such constraints also allow to enforce the (unconditional) presence of a tuple. Parts of the database may be protected from modifications in this way. Note that $\mathbf{C}_0 \subseteq \mathbf{C}_2$, while $\mathbf{C}_1$ and $\mathbf{C}_2$ are incomparable.

Example 2.1. In our example, the schema χ_0 is the tuple $\langle \Psi_0, \Sigma_0 \rangle$, where Ψ_0 consists of the ternary relation symbols *player*, *team*, and *coach*, and Σ_0 can be defined as follows:

$$\begin{aligned}
\sigma_1&: \mathit{player}(x, y, z) \wedge \mathit{player}(x, y', z) \supset y = y',\\
\sigma_2&: \mathit{team}(x, y, z) \wedge \mathit{team}(x, y', z') \supset y = y',\\
\sigma_3&: \mathit{team}(x, y, z) \wedge \mathit{team}(x, y', z') \supset z = z',\\
\sigma_4&: \mathit{coach}(x, y, z) \wedge \mathit{coach}(x, y', z) \supset y = y',\\
\sigma_5&: \mathit{coach}(x, y, z) \wedge \mathit{player}(x', y', z) \supset x \neq x',\\
\sigma_6&: \mathit{coach}(x, y, z) \wedge \mathit{team}(z, y', x') \supset x \neq x'.
\end{aligned}$$

Here σ_1–σ_4 are key constraints, while σ_5 and σ_6 encode that, for any given team, the coach is neither a player nor a team leader. Note that all these constraints are in $\mathbf{C}_0$. □

For a set of relation symbols Ψ, $\mathcal{F}(\Psi)$ denotes the set of all facts $r(\vec{t}\,)$, where $r \in \Psi$ has arity n and $\vec{t} = (c_1, \ldots, c_n) \in \mathcal{U}^n$ is an n-tuple of constants from $\mathcal{U}$. A *database instance* (or simply *database*) for Ψ is any finite set $D \subseteq \mathcal{F}(\Psi)$. The extension of relation r in D is the set of tuples $r^D = \{\vec{t} \mid r(\vec{t}\,) \in D\}$. We denote by $D(\Psi)$ the set of all databases for Ψ. For any relation schema $\chi = \langle \Psi, \Sigma \rangle$, in abuse of notation, $\mathcal{F}(\chi)$ and $D(\chi)$ denote $\mathcal{F}(\Psi)$ and $D(\Psi)$, respectively, and a database for χ is a database for Ψ.

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

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

Example 2.2. Consider the constraint σ_2 in Σ_0, and its ground instance

$$\mathit{team}(\mathit{RM}, \mathit{Roma}, 10) \wedge \mathit{team}(\mathit{RM}, \mathit{Real\ Madrid}, 10) \supset \mathit{Roma}{=}\mathit{Real\ Madrid}.$$

Clearly, this instance does not evaluate true on the relational structure associated with D_0, which therefore violates Σ_0. □

2.2 Datalog$^{\vee,\neg}$ Programs and Queries

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

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \ \mathit{not}\ b_{k+1}, \ldots, \ \mathit{not}\ b_{k+m} \qquad (2)$$

where a_i, b_j are atoms in a relational first-order language $\mathcal{L}$, "*not*" is *negation as failure*, and "," is conjunction. If $k = m = 0$, then ρ is a *fact* and "$\leftarrow$" is omitted. The part left of "$\leftarrow$" is the *head* of ρ, denoted $\mathit{head}(\rho)$, and the part right of "$\leftarrow$" the *body* of ρ, denoted $\mathit{body}(\rho)$. We assume *safety*, i.e., each variable occurring in ρ occurs in some b_i, $1 \leq i \leq k$, whose predicate is not a built-in relation. Built-in relations may occur only in the body.

A *Datalog*$^{\vee,\neg}$ *program* $\mathcal{P}$ is a finite set of Datalog$^{\vee,\neg}$ rules. Important restrictions are *normal programs*, Datalog$^{\neg}$, where $n = 1$ for all rules, *stratified normal* programs, Datalog$^{\neg_s}$, and *nonrecursive* programs, defined as follows. Each Datalog$^{\neg}$ program $\mathcal{P}$ has a *dependency graph* $G(\mathcal{P}) = \langle V, E \rangle$, where V are the predicates occurring in $\mathcal{P}$ and E contains an arc $r \rightarrow s$ if r occurs in $\mathit{head}(\rho)$ and s in $\mathit{body}(\rho)$ for some rule $\rho \in \mathcal{P}$. Moreover, if s occurs under negation, the arc is labeled with '$*$.' Then $\mathcal{P}$ is *stratified*, if $G(\mathcal{P})$ has no cycle with an arc labeled '$*$,' and *nonrecursive*, if $G(\mathcal{P})$ is acyclic.

Semantics. The semantics of a Datalog$^{\vee,\neg}$ program $\mathcal{P}$ is defined via its *grounding* $\mathit{ground}(\mathcal{P})$ w.r.t. $\mathcal{L}$ (usually, the language generated by $\mathcal{P}$), which consists of all ground instances of rules in $\mathcal{P}$ possible with constant symbols from $\mathcal{L}$. Let $B_{\mathcal{L}}$ be the set of all ground atoms with a predicate and constant symbols in $\mathcal{L}$. A *(Herbrand) interpretation for* $\mathcal{P}$ is any subset $I \subseteq B_{\mathcal{L}}$; an atom $p(\vec{c}) \in B_{\mathcal{L}}$ is true in I, if $p(\vec{c}) \in I$, and false in I otherwise. A ground rule (2) is *satisfied* by I, if either some a_i or b_{k+j} is true in I, or some b_i, $1 \leq i \leq k$, is false in I. Finally, I is a model of $\mathcal{P}$, if I satisfies all rules in $\mathit{ground}(\mathcal{P})$.

The *stable model semantics* [Gelfond and Lifschitz 1991] assigns *stable models* to any Datalog$^{\vee,\neg}$ program $\mathcal{P}$ as follows. If $\mathcal{P}$ is "*not*"-free, its stable models are its minimal models, where a model M of $\mathcal{P}$ is minimal, if no $N \subset M$ is a model of $\mathcal{P}$. If $\mathcal{P}$ has negation, M is a stable model of $\mathcal{P}$, if M is a minimal model of the *reduct* $\mathcal{P}$ w.r.t. M, which results from $\mathit{ground}(\mathcal{P})$ by deleting *(i)* each rule ρ with a literal $\mathit{not}\ \ p(\vec{c})$ in the body such that $p(\vec{c}) \in M$, and *(ii)* the negative literals from all remaining rules.

We denote by SM($\mathcal{P}$) the set of stable models of $\mathcal{P}$. 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 [Gelfond and Lifschitz 1991].

Queries. A *Datalog*$^{\vee,\neg}$ *query* Q over a schema $\chi = \langle \Psi, \Sigma \rangle$ is a pair $\langle q, \mathcal{P} \rangle$, where $\mathcal{P}$ is a Datalog$^{\vee,\neg}$ program such that every $p \in \Psi$ occurs in $\mathcal{P}$ only in rule bodies, and q occurs in some rule head of $\mathcal{P}$ but not in Ψ. The *arity of* Q is the arity of q. Given any database D for χ, the *evaluation of* Q *over* D, is $Q[D] = \{(c_1, \ldots, c_n) \mid q(c_1, \ldots, c_n) \in M, \text{ for each } M \in SM(\mathcal{P} \cup D)\}$. Note that as for Q, any nonrecursive $\mathcal{P}$ can be rewritten to a *union of conjunctive queries*, that is, a set of rules (2) where $n = 1$ and $m = 0$, with the same head predicate q which does not occur in rule bodies. For further background on Datalog$^{\vee,\neg}$ and queries, see Abiteboul et al. [1995] and Eiter et al. [1997].

Example 2.3. In our ongoing example, we may consider a query Q that asks for the codes of all players and team leaders, and that is formally written as $Q = \langle q, \mathcal{P} \rangle$ where $\mathcal{P} = \{q(x) \leftarrow \mathit{player}(x, y, z), q(x) \leftarrow \mathit{team}(v, w, x)\}$. Q has arity 1. Note that $\mathcal{P}$ is a union of conjunctive queries.

3. CONSISTENT QUERY ANSWERING FRAMEWORK

3.1 A General Framework for Database Repairs

Let us assume that $\chi = \langle \Psi, \Sigma \rangle$ is given together with a (possibly inconsistent) database D for χ. Following a common approach in the literature on inconsistent databases [Arenas et al. 1999; Greco et al. 2003; Calì et al. 2003a; Chomicki 2007], we next define the semantics of querying D in terms of its *repairs*. Specifically, we present a generalization of previous approaches where the way of repairing a database is chosen according to an arbitrary preorder on databases satisfying some conditions.

We suppose that $\leq_D$ is a (fixed) preorder (i.e., a reflexive and transitive binary relation) on $D(\chi)$, and denote by $<_D$ the induced preference order (i.e., an irreflexive and transitive binary relation) given by $R_1 <_D R_2$, if $R_1 \leq_D R_2 \wedge R_2 \not\leq_D R_1$. We call R_1 $<_D$*-preferred* to R_2 in this case. A repair for D is now defined in terms of a minimal element under $<_D$.

Definition 3.1 (*Repair*). Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, and let $\leq_D$ be a (fixed) preorder on $D(\chi)$. A database $R \in D(\chi)$ is a $\leq_D$*-repair* (simply, *repair*) *for D w.r.t.* χ, if

(1) $R \models \Sigma$, and

(2) R is minimal in $D(\chi)$ w.r.t. $<_D$, i.e., there is no $R' \in D(\chi)$ such that $R' \models \Sigma$ and R' is $<_D$-preferred to R.

The set of all repairs for D w.r.t. χ is denoted by $rep_{\chi}^{\leq_D}(D)$. When clear from the context, the subscript χ may be dropped. Similarly, the superscript $\leq_D$ is omitted.

The definition of repair relies on a general notion of preorder on databases. The method for consistent query answering presented in the following is based on abstract properties of the induced preference order, which we refer to as set inclusion proximity, disjoint preference expansion and disjunctive split. The property of *set inclusion proximity* is as follows:

(SIP) For any databases R_1, R_2, and D, $\triangle(R_1, D) \subset \triangle(R_2, D)$ implies $R_1 <_D R_2$,

where $\triangle(A, B) = (A \setminus B) \cup (B \setminus A)$ is symmetric set difference. Informally, this property effects that a database R satisfying the constraints can be a repair only if there is no way to establish consistency with Σ by touching merely a strict subset of facts compared to R.

The properties *disjoint preference expansion* and *disjunctive split* are as follows:

(DPE) If $R_1 <_{D_1} R'_1$ and R_2, D_2 are disjoint from R_1, R'_1, and D_1 (i.e., $(R_1 \cup R'_1 \cup D_1) \cap (R_2 \cup D_2) = \emptyset$), then $R_1 \cup R_2 <_{D_1 \cup D_2} R'_1 \cup R_2$.

(DIS) If $R_1 <_D R_2$, then for every database R it holds that either $R_1 \cap R <_{D \cap R} R_2 \cap R$ or $R_1 \setminus R <_{D \setminus R} R'_1 \setminus R$ (or both).

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

The prototypical preorder $\leq_D$ is given by $R_1 \leq_D R_2$ iff $\triangle(R_1, D) \subseteq \triangle(R_2, D)$ [Arenas et al. 1999, 2003; Barceló and Bertossi 2003; Bravo and Bertossi 2003; Chomicki et al. 2004a; Greco et al. 2003; Fuxman and Miller 2007]. Intuitively, each repair of D is then obtained by properly adding and deleting facts from D in order to satisfy constraints in Σ, as long as we "minimize" such changes. The following proposition is easy to prove.

PROPOSITION 3.1. *The prototypical preorder satisfies properties (SIP), (DPE), and (DIS).*

In our examples we refer to the prototypical preorder. 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, beside those based on the prototypical preorder previously discussed, including set-inclusion based ordering [Fagin et al. 1983; Calì et al. 2003a], cardinality-based ordering [Arenas et al. 2003; Lin and Mendelzon 1998], weight-based orderings [Lin 1996], as well as refinements with priority levels. An interesting special case of weight-based ordering is the lexicographic preference, where R_1 is preferred to R_2 w.r.t. D if the first fact in a total ordering of $\mathcal{F}(\chi)$ on which R_1 and R_2 repair D differently belongs to R_2. However, we point out that our methods and results for query answering can also be extended to other preference orderings under certain conditions (see Section 9).

3.2 Constructible Repairs and Safe Constraints

An important aspect is that constraints might enforce that *any* set of facts R for $\chi = \langle \Psi, \Sigma \rangle$ which satisfies Σ must be infinite, and thus χ is inconsistent, that is, no $D \in D(\chi)$ satisfies Σ. A simple example is where $\Sigma = \{\forall x\, p(x)\}$. Semantically, this is commonly avoided by requesting domain-independence of constraints

[Ullman 1989], which syntactically is ensured by *safety*; that is, each variable occurring in the head of a constraint must also occur in its body. Notice that major classes of constraints including key constraints, functional dependencies, exclusion dependencies, inclusion dependencies of the form $p_1(\vec{x}) \supset p_2(\vec{x})$, or denial constraints fulfill safety. Together with (SIP), safety of constraints ensures that any database D has a repair if this is possible at all. For any $R \subseteq \mathcal{F}(\chi)$, we denote by $adom(R, \chi)$ the *active domain* of R and χ, i.e., the set of constants occurring in R and Σ.

Proposition 3.2. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, where all constraints in Σ are safe. Suppose that $<_D$ satisfies (SIP). Then, every repair $R \in rep(D)$ involves only constants from $adom(D, \chi)$, and some repair exists if χ is consistent.*

Notice that, for an arbitrary preference order, no repair may exist, even if χ is consistent. In the rest of this article, unless stated otherwise, we assume safe constraints.

Finite repairs can also be ensured for unsafe constraints in which variables violating safety are guarded by built-in relations, such as for $D = \emptyset$ w.r.t. $\chi = \langle \{p\}, \{p(x) \vee x > 100\} \rangle$, assuming that $\mathcal{U}$ are the natural numbers. As this example shows, repairs may go beyond the active domain. However, this is prevented if built-ins involve only equality and inequality. We have a result similar to Proposition 3.2 (cf. Appendix A for proofs).

Proposition 3.3. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$ where no built-in relations occur in Σ except $=$ and $\neq$. Suppose that $<_D$ satisfies (SIP). Then, every repair $R \in rep(D)$ involves only constants from $adom(D, \chi)$, and some repair exists if χ is consistent.*

3.3 Queries and Consistent Answers

The notion of repair is crucial for the definition of the semantics of querying inconsistent databases. We conclude this section by formalizing this aspect.

Definition 3.2. Let Q be a nonrecursive Datalog$^\neg$ query. For any database $D \in D(\chi)$, the set of *consistent answers to Q w.r.t. D* is the set of tuples $ans_c(Q, D) = \{\vec{t} \mid \vec{t} \in Q[R], \text{ for each } R \in rep_\chi(D)\}$. □

Informally, a tuple $\vec{t}$ is a consistent answer if it is a consequence under standard certainty semantics for each possible repair of the database D. Note that in real applications, a query language subsumed by nonrecursive Datalog$^\neg$ is often adopted.

Example 3.1. Recall that in our scenario, repairs for the database D_0 for χ_0 are shown in Figure 1. 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 $ans_c(Q, D_0) = \{(8), (9), (10)\}$. For the query $Q' = \langle q, \{q(y) \leftarrow team(x, y, z)\} \rangle$, we have $ans_c(Q', D_0) = \{(Barcelona)\}$, while for $Q'' = \langle q', \{q'(x, z) \leftarrow team(x, y, z)\} \rangle$, we have $ans_c(Q'', D_0) = \{(RM, 10), (BC, 8)\}$. □

4. LOCALITY PROPERTIES FOR REPAIRING INCONSISTENT DATABASES

In this section, we investigate how to localize inconsistency in a given database D, that is, how to narrow down the set of facts in D to a part which is "affected" by inconsistency and repair, and how to obtain the repairs of D from the repairs of this affected part. To this end, we introduce the notion of a *repair envelope*. Informally, a repair envelope is a set of facts E such that the repairs of D touch only facts in E and are given by the repairs of $D \cap E$ plus the "unaffected" ("safe") part of D, that is, the portion of D which is outside the envelope. More formally:

Definition 4.1. Let D be a database for a relational schema $\chi = \langle \Psi, \Sigma \rangle$. A set E of facts over Ψ is a repair envelope for D if it fulfills the following conditions:

$$\triangle(R, D) \subseteq E, \quad \text{for all } R \in rep_{\chi}(D), \tag{3}$$

$$rep_{\chi}(D) = \{R \cup (D \setminus E) \mid R \in rep_{\chi}(D \cap E)\}. \tag{4}$$

The repairs of D can then be fully localized to the repairs of $D \cap E$, which in practice may be much smaller than D. In fact, as will be shown in this section, for constraints $\mathbf{C}_0$ the set of all facts involved in constraint violations, denoted C (formally defined in the beginning of Section 4.1), is always a repair envelope, and for constraints $\mathbf{C}_1$ and $\mathbf{C}_2$, a closure C^* of C under syntactic conflict propagation, that is, under co-occurrence of facts in violated constraints (cf. Definition 4.4), is a repair envelope. Such a closure, as we will explain in detail in the following, takes care of facts that "indirectly" participate in constraint violations. In general, however, a repair envelope needs not be a superset of C^* or C. Figure 2 shows the different sets.

Example 4.1. Recall that $team(RM, Roma, 10) \wedge team(RM, Real\ Madrid, 10) \supset Roma = Real\ Madrid$ witnesses in Example 2.2 a violation of the key of *team*; it is the only ground constraint violated by D_0. Here, $C = \{team(RM, Roma, 10), team(RM, Real\ Madrid, 10)\}$, and since the constraints are of type $\mathbf{C}_0$, it is a repair envelope for D. The database $D \cap C = C$ has the two repairs $R_1 = \{team(RM, Roma, 10)\}$ and $R_2 = \{team(RM, Real\ Madrid, 10)\}$; therefore, according to (4), D has the two repairs $R_1 \cup (D_0 \setminus C)$ and $R_2 \cup (D_0 \setminus C)$, which are those shown in Figure 1. □

The following example shows that taking only C into account is not always sufficient.

Example 4.2. Let $D = \{s(a)\}$ for $\chi = \langle \Psi, \Sigma \rangle$, where $\Sigma = \{s(a) \supset r(a), r(a) \supset p(a) \vee q(a), r(a) \wedge p(a) \supset a \neq a\}$. In this case $C^* = \{s(a), r(a), p(a), q(a)\}$ and the constraints are of type $\mathbf{C}_1$, hence, C^* is a repair envelope for D. The database $D \cap C^* = D$ has the two repairs $R_1 = \emptyset$ and $R_2 = \{s(a), r(a), q(a)\}$. Note that $\triangle(R_2, D) = \{r(a), q(a)\} \nsubseteq \{s(a), r(a)\} = C$. Therefore, C violates (3) and is not a repair envelope.

Note that a repair envelope always exists, since the set of all facts is a trivial repair envelope. As for localizing the computation of $rep(D)$, only Condition (4) is relevant (if E satisfies it, then so does every E' such that $E' \cap D = E \cap D$, in particular $E' = E \cap D$). Condition (3), however, allows to bound the answer to certain queries. In particular, for monotone queries Q, we have that $Q[D \setminus E] \subseteq ans(Q, D) \subseteq Q[D \cup E]$.

For general constraints, C^* is not always a repair envelope. However, we show that it is a *weak repair envelope*:

Definition 4.2. Let D be a database for a relational schema $\chi = \langle \Psi, \Sigma \rangle$. A set E of facts over Ψ is a weak repair envelope for D if it fulfills Condition (3) and instead of (4), the relaxed equation

$$rep_{\chi}(D) = \{(R \cap E) \cup (D \setminus E) \mid R \in rep_{\chi}(D \cap E)\}. \tag{5}$$

That is, the repairs of D are obtained by constraining the repairs of $D \cap E$ to the repair envelope. This is necessary since facts outside the envelope might be added to such repairs (see Example 4.4). However, this can only occur in presence of certain disjunctions.

Despite the difference that E is either a repair envelope or a weak repair envelope, we define affected database and safe database w.r.t. E as follows.

Definition 4.3. Let D be a database for a relational schema $\chi = \langle \Psi, \Sigma \rangle$ and E a (weak) repair envelope. Then $D \cap E$ is the affected part of D (w.r.t. χ), or simply the "affected database", and $D \setminus E$ is the safe part of D (w.r.t. χ), or simply the "safe database". □

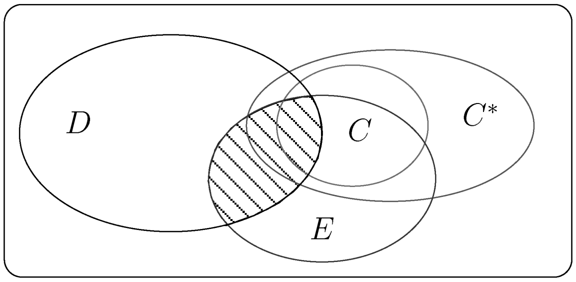
Note that we mostly refer to specific kinds of envelopes in the paper, namely the conflict set C and the conflict closure C^*. We proceed as follows. After formally defining C and C^* and establishing some auxiliary results, we show that C^* is a weak repair envelope in general. We then prove that it is a repair envelope under restrictions, in particular for $\mathbf{C}_1$ and $\mathbf{C}_2$ constraints. This envelope may be further decreased. Indeed, we prove that C is a repair envelope for $\mathbf{C}_0$ constraints. In fact, the results for special constraints are stronger and establish 1-1 correspondences between repairs of $D \cap E$ and repairs of D. Some of the proofs are omitted here but can be found in Appendix B.

4.1 General Constraints

Let D be a database for a relational schema $\chi = \langle \Psi, \Sigma \rangle$. The *conflict set* for D w.r.t. χ is the set of facts $C_{\chi}(D) = \{p(\vec{t}\,) \mid \exists \sigma \in ground(\Sigma),\ p(\vec{t}\,) \in facts(\sigma),\ D \not\models \sigma\}$, i.e., $C_{\chi}(D)$ is the set of facts occurring in the ground instances of Σ which are violated by D. In the following, if clear from the context, D and/or the subscript χ will be dropped.

Figure 2 shows that the conflict set may contain both facts in D (as in Example 4.1) and facts in $\mathcal{F}(\chi)$ that do not belong to D. For example, let $D = \{p(a)\}$, and let χ contain the dependency $p(x) \supset q(x)$. Then $C = \{p(a), q(a)\}$.

For defining conflict propagation, we first introduce the following notion. Two facts $p(\vec{t}\,)$, $p'(\vec{t}\,')$ in $\mathcal{F}(\chi)$ are *constraint-bounded in* χ, if there exists some $\sigma \in ground(\Sigma)$ such that all constants occurring in $facts(\sigma)$ are from $adom(D, \chi)$, and $\{p(\vec{t}\,), p'(\vec{t}\,')\} \subseteq facts(\sigma)$. (Note that by assumed safety of constraints and the results of Section 3.2, we only need to consider $adom(D, \chi)$.) We now generalize the notion of conflict set.



Conflict set C : facts occurring in $ground(\Sigma)$ violated in D
Conflict closure C^* : syntactic conflict propagation from C by Σ
Repair envelope E : safe bound on tuple changes with local repairs (hatched)

Fig. 2. Localization of database repair.

Definition 4.4 (*Conflict Closure*). Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$. Then, the *conflict closure* for D, denoted by $C^*_\chi(D)$, is the least set $A \supseteq C_\chi(D)$ which contains every fact $p(\vec{t}\ )$ constraint-bounded in χ with some fact $p'(\vec{t}\,') \in A$.

We omit D and/or the subscript χ if clear from the context. Intuitively, C^* contains, besides facts from C, facts which possibly must be touched by repair in turn to avoid new inconsistency with Σ caused by previous repairing actions. For example, assume that χ contains the constraints $p(x) \supset q(x)$ and $q(x) \supset s(x)$. Then, for $D = \{p(a)\}$, we have that $C = \{p(a), q(a)\}$ and $C^* = C \cup \{s(a)\}$. As shown in Figure 2, C^* may add to C both facts inside and outside D, but may also coincide with C, as in our example if $s(a)$ was in D.

Towards a proof that C^* is a weak repair envelope, we need some preliminary technical results. For D and $\chi = \langle \Psi, \Sigma \rangle$, consider the following two sets of ground constraints:

(*i*) $\Sigma^a_\chi(D) = \{\sigma \in ground(\Sigma) \mid facts(\sigma) \cap C^* \neq \emptyset\}$ consists of all ground constraints in which at least one fact from C^* occurs;
(*ii*) $\Sigma^s_\chi(D) = \{\sigma \in ground(\Sigma) \mid facts(\sigma) \not\subseteq C^*\}$ consists of all ground constraints in which at least one fact occurs which is *not* in C^*.

As usual, χ and/or D will be omitted. We first show that $\Sigma^a \cup \Sigma^s$ is a special partitioning of $ground(\Sigma)$.

Proposition 4.1 (Separation). *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$. Then, (1) $facts(\Sigma^a) = C^*$, (2) $facts(\Sigma^s) \cap C^* = \emptyset$, (3) $\Sigma^a \cap \Sigma^s = \emptyset$, and (4) $\Sigma^a \cup \Sigma^s = ground(\Sigma)$.*

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

Proposition 4.2 (Safe Database). *Let D be any database for $\chi = \langle \Psi, \Sigma \rangle$. Then, for each repair $R \in rep(D)$ it holds that $R \setminus C^* = D \setminus C^*$.*

Informally, the above proposition shows that $D \setminus C^*$ is a safe portion of D, in the sense that tuples of D outside the conflict closure will not be touched by repair.

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

LEMMA 4.3. *Let D be a database for $\chi = \langle\Psi, \Sigma\rangle$, and let $A = D \cap C^*$ and $\chi^a = \langle\Psi, \Sigma^a\rangle$. Then, for each $S \subseteq D \setminus C^*$, the following holds:*

(1) *for each $R \in rep_\chi(A \cup S)$, it holds that $R \cap C^* \in rep_{\chi^a}(A)$;*
(2) *for each $R \in rep_{\chi^a}(A)$ there exists a set of facts $S' \subseteq \mathcal{F}(\chi)$ such that $S' \cap C^* = \emptyset$, and $(R \cup S') \in rep_\chi(A \cup S)$.*

In other words, item (1) in the lemma above shows how to obtain a repair of the database $A = D \cap C^*$ w.r.t. χ^a, from a repair, computed w.r.t. χ, of A augmented with any subset S of the safe database $D \setminus C^*$. Conversely, item (2) shows how to obtain a repair of $A \cup S$ w.r.t. χ, from a repair of A w.r.t. χ^a. Notice that repairing A w.r.t. χ^a, and not w.r.t. χ, is necessary for the lemma above to hold, since for a repair $R \in rep_\chi(A \cup S)$, it does not hold in general that $R \cap C^* \in rep_\chi(A)$. Also, repairing A w.r.t. χ^a avoids repairing constraints in Σ^s not satisfied by A.

Armed with the above concepts and results, we state one of the main theorems of this section.

THEOREM 4.4. *Every database D for $\chi = \langle\Psi, \Sigma\rangle$ has C^* as a weak repair envelope.*

PROOF. We first show that for each $R \in rep_\chi(D)$ then $\triangle(R, D) \subseteq C^*$, as specified by condition (3) in Section 4, where we pose $E = C^*$. Assume by contradiction that there exists a fact $f \in \triangle(R, D)$ such that $f \notin C^*$. By Proposition 4.1 it follows that there exists no $\sigma \in \Sigma^a$ such that $f \in facts(\sigma)$. Then, if $f \in R \setminus D$, it is easy to see that $R \setminus \{f\} \models \Sigma$, but by property (SIP) we have that $R \setminus \{f\} <_D R$, thus contradicting the assumption that $R \in rep_\chi(D)$. Analogously, if $f \in D \setminus R$, it is easy to see that $R \cup \{f\} \models \Sigma$, but by property (SIP) we have that $R \cup \{f\} <_D R$, thus again contradicting the assumption.

We now prove that $rep_\chi(D)$ coincides with the set defined by Equation (5) in Section 4, where we pose $E = C^*$. To this aim, we show that (i) for every $R \in rep_\chi(D)$, there exists some $R' \in rep_\chi(D \cap C^*)$ such that $R = (R' \cap C^*) \cup (D \setminus C^*)$, and (ii) for every $R \in rep_\chi(D \cap C^*)$ there exists some $R' \in rep_\chi(D)$ such that $R' = (R \cap C^*) \cup (D \setminus C^*)$.

(i) We first apply Item 1 of Lemma 4.3 for $S = D \setminus C^*$ and obtain $R \cap C^* \in rep_{\chi^a}(D \cap C^*)$ (notice that in Lemma 4.3 $A = D \cap C^*$, $\chi^a = \langle\Psi, \Sigma^a\rangle$, and since we pose $S = D \setminus C^*$, we have that $A \cup S = D$). We then apply Item 2 for $S = \emptyset$, and we obtain that there exists S' such that $S' \cap C^* = \emptyset$ and $R' = (R \cap C^*) \cup S' \in rep_\chi(A)$. Since $S' \cap C^* = \emptyset$, we also have that $R' \cap C^* = R \cap C^*$, and from Proposition 4.2, it follows that $R = (R \cap C^*) \cup (D \setminus C^*)$. Therefore, $R = (R' \cap C^*) \cup (D \setminus C^*)$.

(ii) Similarly, applying first Item 1 of Lemma 4.3 for $S = \emptyset$ and then Item 2 for $S = D \setminus C^*$, we obtain that there exists S' such that $S' \cap C^* = \emptyset$, and $R' = (R \cap C^*) \cup S' \in rep_\chi(D)$. Then, from Proposition 4.2, it follows that $S' = D \setminus C^*$. We thus easily obtain that $R' = (R \cap C^*) \cup (D \setminus C^*)$. □

For computing repairs for an inconsistent database D, we can thus proceed as follows:

1. compute the conflict closure C^* (w.r.t. Σ);
2. compute the repairs of $A = D \cap C^*$ (w.r.t. Σ);
3. intersect each repair obtained with C^*; and
4. for each such set, take the union with $D \setminus C^*$.

In fact, as we will show, it is sufficient to consider Σ^a in Step 2 instead of Σ. We note that C^* is polynomially computable w.r.t. data volume from C by transitive closure computation (simply use a Datalog program over C), but omit discussing efficient algorithms.

A drawback of the approach in general, however, is that in Step 2, facts outside C^* might be included in a repair of A, which are stripped off subsequently in Step 3.

Example 4.3. Consider $D = \{p(a)\}$ for $\chi = \langle \Psi, \{p(a), q(a)\}\rangle$. In this case, $C = C^* = \{q(a)\}$, $A = D \cap C^* = \emptyset$, and $D \setminus C^* = D$. We have $rep(A) = \{\{p(a), q(a)\}\}$ and $\{p(a), q(a)\} \cap C^* = \{q(a)\}$; $p(a)$ is stripped off from the repair of A.

In this example, the repair of A added a fact outside C^* but from the safe part of D, which doesn't hurt. The following example shows that facts outside $C^* \cup D$ may be added.

Example 4.4. Consider $D = \{r(a), p(a)\}$ for $\chi = \langle \Psi, \Sigma\rangle$, where $\Sigma = \{r(a) \supset p(a) \vee q(a),\ r(a),\ s(a)\}$. Then, $C^* = \{s(a)\}$ and $D \cap C^* = \emptyset$ has two repairs, viz. $R_1 = \{r(a), s(a), p(a)\}$ and $R_2 = \{r(a), s(a), q(a)\}$. Note that if C^* were a repair envelope, then according to Condition (4), $R_2 \cup (D \setminus C^*) = \{r(a), s(a), p(a), q(a)\}$ would have to be a repair of D, which is incorrect. Note also, that this time $q(a)$ has been added in R_2 which was neither in D nor in C^*.

We remark that in Example 4.4, Σ contains constraints from both the classes $\mathbf{C}_1$ and $\mathbf{C}_2$, but not from a single class. As we show in the next subsection, the effects in Example 4.4 can not happen under restriction to a single class, and C^* is always a repair envelope.

We finally provide the result below that follows from Theorem 4.4, and remarks that repairing basically depends on Σ^a.

COROLLARY 4.5. *Let D be a database for $\chi = \langle \Psi, \Sigma\rangle$, and let $\chi' = \langle \Psi, \Sigma'\rangle$ be such that $\Sigma^a_{\chi'}(D) = \Sigma^a_{\chi}(D)$. Then $rep_{\chi}(D) = rep_{\chi'}(D)$.*

Then, we can modify or prune constraints "outside" Σ^a in arbitrary manner, e.g., for optimization purposes. As we show in the next subsection, this makes C^* a repair envelope, rather than a weak repair envelope, in several cases in which Σ contains general constraints.

4.2 Special Constraints

In this section, we consider the constraint classes $\mathbf{C}_i$ which have been introduced in Section 2, and determine repair envelopes for them.

4.2.1 *Constraints* $\mathbf{C}_1$ *and* $\mathbf{C}_2$. Recall that $\mathbf{C}_1$ constraints have nonempty bodies, and thus cannot unconditionally enforce the inclusion of facts to a database instance.

Proposition 4.6. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$ such that $\Sigma \subseteq \mathbf{C}_1$. Then, each repair R of $A = D \cap C^*$ w.r.t. χ satisfies $R \subseteq C^*$.*

Proof. By Item 1 of Lemma 4.3, for $S = \emptyset$, each $R \in rep_\chi(A)$ gives rise to a repair $R' = R \cap C^*$ of A w.r.t. $\chi^a = \langle \Psi, \Sigma^a \rangle$. By Item 2 of Lemma 4.3, for $S = \emptyset$, R' in turn gives rise to a repair R'' of A w.r.t. χ of the form $R'' = R' \cup S'$ such that $S' \cap C^* = \emptyset$. Since clearly $S' \models \Sigma^s$, property (DPE) implies that S' is a repair of $S = \emptyset$ w.r.t. $\langle \Psi, \Sigma^s \rangle$. Since each constraint in Σ^s has a nonempty body, it follows by (SIP) that $S' = \emptyset$. Hence $R \cap C^*$ is a repair of A w.r.t. χ. Now if $R \not\subseteq C^*$ held, then $\triangle(R'', A) \subset \triangle(R, A)$ would hold, which by (SIP) implies $R'' <_D R$. This is a contradiction. □

Recall that $\mathbf{C}_2$ are the nondisjunctive constraints; that is, every constraint has at most one database atom in the head.

Proposition 4.7. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, where $\Sigma \subseteq \mathbf{C}_2$. Then (i) every repair R of $A = D \cap C^*$ satisfies $R \subseteq D \cup C^*$, and (ii) for every repairs R, R' of A, $R \cap (D \setminus C^*) = R' \cap (D \setminus C^*)$.*

Proof. By the argument in the proof of Proposition 4.6, every $R \in rep(A)$ gives rise to some $R'' \in rep(A)$ of the form $R'' = (R \cap C^*) \cup S'$ such that $S' \cap C^* = \emptyset$ and S' is a repair of $S = \emptyset$ w.r.t. $\langle \Psi, \Sigma^s \rangle$. As each constraint in Σ^s is nondisjunctive, there is the least (w.r.t. $\subseteq$) set of facts $\mathcal{F}$ such that $\mathcal{F} \models \Sigma^s$ (in essence, Σ^s is a Horn theory), and $\mathcal{F} \subseteq S'$ must hold; by (SIP), $\mathcal{F} = S'$. Now if $R \not\subseteq C^* \cup D$ held, then $\triangle(R'', A) \subset \triangle(R, A)$ would hold ($\mathcal{F} \subseteq R$ must hold, and thus $R'' \subseteq R$), which by (SIP) means $R'' <_D R$. This is a contradiction, and proves (i). Item (ii) holds as $R \cap (D \setminus C^*) = \mathcal{F}$ for each $R \in rep(A)$. □

The proposition above allows us to exploit Theorem 4.4 in a constructive way for many significant classes of constraints, for which it implies a bijection between the repairs of a database D, and the repairs of the affected part $A = D \cap C^*$.

Corollary 4.8. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$ where $\Sigma \subseteq \mathbf{C}_i$, for $i \in \{1, 2\}$. Then, C^* is a repair envelope for D. In fact, there exists a bijection $\mu : rep(D) \to rep(D \cap C^*)$, such that for every $R \in rep(D)$, $R = \mu(R) \cup (D \setminus C^*)$.*

By this result, the repairs of a database D can be computed by avoiding Step 3 of the procedure given in Section 4.1. Note also that by the above corollary and Proposition 4.1 and Corollary 4.5, we can make C^* a repair envelope for an arbitrary relational schema $\chi = \langle \Psi, \Sigma \rangle$, if we can modify Σ to constraints Σ' from $\mathbf{C}_1$ or $\mathbf{C}_2$ while preserving the affected constraints, that is, $\Sigma^a_\chi(D) = \Sigma^a_{\langle \Psi, \Sigma' \rangle}(D)$. Technically, this can be exploited in different ways, for example, by dropping constraints, adding ground instances of constraints, rewriting constraints by modifying the built-in part (in fact, only semantic equivalence of affected ground constraints is needed), etc.

We also remark that C^* may be decreased to a smaller repair envelope, by taking tuple generating constraints into account. For example, if $p(a)$ belongs to each repair (e.g., enforced by a constraint), $p(a)$ can be removed from the repair envelope. If there is another constraint $p(x) \supset q(x)$, also $q(a)$ can be removed. Exploring this is left for further study.

4.2.2 *Constraints* $\mathbf{C}_0$. Recall that constraints in $\mathbf{C}_0$ have only built-in relations in the head. Notably, the repairs of a database with integrity constraints from this class are computable by focusing on the immediate conflicts in the database, without the need of computing the conflict closure set, which may be onerous in general. Furthermore, repairs always do only remove tuples from relations, but never include new tuples. We will next formally prove these properties, starting with the following proposition.

PROPOSITION 4.9. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, $\Sigma \subseteq \mathbf{C}_0$, and let $A = D \cap C^*$. Then,*

(1) *$C \subseteq D$;*

(2) *for each $R \in rep(A)$, (i) $R \subseteq A$, (ii) $\triangle(R, A) \subseteq C$, (iii) $A \backslash C \subseteq R$, and (iv) $R \cap C \in rep(C)$;*

(3) *for each $R \in rep(C)$, $R \cup (A \setminus C) \in rep(A)$.*

Note that Proposition 4.9 shows that each repair of the conflict set C just removes tuples from C (take $D = C$ in Item 2.(ii)). Furthermore, because $C \subseteq D$, we can compute C efficiently by suitable SQL statements which express constraint violations. We are now ready to prove that under $\mathbf{C}_0$ constraints, we can use C instead of C^* as a repair envelope, and thus avoid the onerous construction of C^*. In fact, we prove a more general result.

THEOREM 4.10. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$ where $\Sigma \subseteq \mathbf{C}_0$. Then, every set of facts $E \supseteq C$ is a repair envelope for D. Moreover, there exists a bijection $\nu : rep(D) \rightarrow rep(D \cap E)$, such that for each $R \in rep(D)$, $R = \nu(R) \cup (D \setminus E)$.*

PROOF. By Corollary 4.8, there is a bijection $\mu : rep(D) \rightarrow rep(A)$, where $A = D \cap C^*$, such that the repairs of D are given by $\mu(R) \cup (D \setminus C^*)$, for all $R \in rep(A)$. Items 1 and 2.(iv) of Proposition 4.9 and the fact that each repair R of C satisfies $R \subseteq C$, imply that all repairs of A are given by $(A \setminus C) \cup R$, where $R \in rep(C)$. Hence, the mapping $\nu : rep(D) \rightarrow rep(C)$ given by $\nu(R) = \mu(R) \cap C$ is a bijection such that

$$\begin{aligned} R &= \mu(R) \cup (D \setminus C^*) \\ &= \nu(R) \cup (A \setminus C) \cup (D \setminus C^*) \\ &= \nu(R) \cup ((D \cap C^*) \setminus C) \cup (D \setminus C^*) \;=\; \nu(R) \cup (D \setminus C) \end{aligned}$$

This proves the result for $E = C$. For general $E \supseteq C$, we note that $D' = D \cap E$ and D have the same conflict set; hence, there exists a bijection $\nu' : rep(D \cap E) \rightarrow rep(C)$ such that $R = \nu'(R) \cup (D' \setminus C)$, for each $R \in rep(D')$. This implies a bijection $\nu'' : rep(D) \rightarrow rep(D \cap E)$ of the given form. □

Consequently, in this setting we can compute the repairs of a database D as follows:

1. compute C,
2. compute the repairs R of C (where $R \subseteq C \subseteq D$), and
3. take for each such repair R the union with $D \setminus C$.

An example of application of the above procedure has been given in Example 4.1. The fact that every $E \supseteq C$ is a repair envelope gives convenient flexibility to modify the SQL statements for computing C (i.e., one may sensibly simplify conditions in the SQL statements such that they are easier to evaluate but might infer more tuples).

5. QUERY ANSWERING THROUGH LOCALIZED REPAIRS

The localization properties discussed in the previous section may be used to optimize consistent query answering from an inconsistent database D. Indeed, based on them, one may conceive an optimization procedure consisting of the following three steps:

Focusing Step. Localize inconsistency in D, and single out facts that are affected by repair, and facts that are not, that is, compute the (weak) repair envelope E and the affected database $D \cap E$ and the safe database $D \setminus E$.

Decomposition Step. Compute repairs of the affected database, and obtain from them repairs of D (by suitably incorporating the safe database).

Recombination Step. Recombine the repairs of D for computing the consistent query answers, i.e., evaluate the query over local repairs augmented with the safe database and compose consistent query answers from these query results.

In situations in which the size of the affected database is much smaller than the size of the database D, computing the repairs of the affected database is significantly faster than the naive computation, which just aims at changing tuples “randomly” in the database, and does not in general rely on a focusing strategy. Moreover, localizing the inconsistency can be carried out easily by evaluating the constraints issued over the schema (by means of suitable SQL statements). Focusing and decomposition have been amply discussed in Section 4. In this section, we address the issue of efficient recombination, by illustrating three (increasingly elaborate) approaches for its implementation:

—A basic methodology for evaluating the query over local repairs augmented with the safe database is discussed in Section 5.1. This methodology linearly scales w.r.t. the number of repairs, but possibly exponentially w.r.t. the size of the affected database. Yet, it can be applied to arbitrary queries and constraints.

—A more elaborate approach is then discussed in Section 5.2, based on the concept of *repair factorization*. Roughly, it aims at decomposing a repair envelope into disjoint components such that the repairs of D can be efficiently obtained from their repairs. Note that here two aspects are crucial:

(i) The ability to identify such components. We propose sufficient conditions for factorization, based on which components can be identified in polynomial time for practical relevant classes of constraints (e.g., $\mathbf{C}_0$ and $\mathbf{C}_1$).

(*ii*) The ability to combine the component repairs. We define in Section 5.2.1 two kinds of relevant components: *singular* components, which though inconsistent can be repaired in one particular rather than in all the possible ways to compute consistent answers to a certain query Q; and *decomposable* components, whose repairs can be processed in an independent (parallel) manner. When all the components of a factorization fall in one of these two categories (which can be efficiently checked by analyzing both Q and D), consistent query answering is polynomial.

—Eventually, in Section 5.2.2 we show how to combine repair factorization with other techniques. In particular, we present a *query grounding* strategy which reduces a general query to a set of ground queries with the effect that more components can become singular while decomposability of components is unaffected. We thus enlarge the class of queries for which consistent answers are computable in polynomial time, and in fact single out a purely syntactic condition for tractability.

Towards the combined application of these techniques, we may want to implement recombination via the repair factorization approach for constraints in $\mathbf{C}_0$ or $\mathbf{C}_1$, for instance. If some nonsingular component emerges, then we may exploit the query grounding strategy, provided that all other components are decomposable. If some component remains neither singular nor decomposable, then we eventually have to resort to the basic method.

5.1 Recombination Step

Let us now consider the problem of evaluating a query Q issued over an inconsistent database D for χ, i.e., to compute $ans_c(Q, D)$. Recall that according to the definition in Section 2, a tuple $\vec{t}$ belongs to $ans_c(Q, D)$ if $\vec{t}$ is in the evaluation of Q over every repair of D, i.e., $ans_c(Q, D) = \{\vec{t} \mid \vec{t} \in Q[R] \text{ for each } R \in rep(D)\} = \bigcap_{R \in rep(D)} Q[R]$. The following proposition, which is immediate from the definitions, states how we can exploit repair envelopes for localization in query answering.

Proposition 5.1. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, and let Q be a nonrecursive Datalog$^{\neg}$ query. Let E be a set of facts, and let $A = D \cap E$ and $S = D \setminus E$. Then*

$$ans_c(Q, D) = \bigcap_{R \in rep(A)} Q[\varepsilon(R) \cup S], \qquad (6)$$

where (i) $\varepsilon(R) = R$ if E is a repair envelope for D, and (ii) $\varepsilon(R) = R \cap E$ if E is a weak repair envelope for D.

By the results from above, we can always apply (ii) of (6) with $E = C^*$, and for $\mathbf{C}_1$ or $\mathbf{C}_2$ constraints apply always (i) of (6) with $E = C^*$. Furthermore, for $\mathbf{C}_0$, we can apply always (i) of (6) with $E = C$. Since in this case $C \subseteq D$, we can rewrite (6) to $ans_c(Q, D) = \bigcap_{R \in rep(C)} Q[R \cup S]$.

In the light of the equations above, query answering can be carried out by "locally" repairing the affected database, and evaluating the query over each local repair augmented with the safe portion of the data. While this approach has the advantage of localizing the inefficient repair computation (repair checking is already co-NP-hard, cf. Chomicki [2007] and references therein) on a fragment A of the database D, its implementation leads to an algorithm for consistent query answering which linearly scales w.r.t. the number of repairs, but possibly exponentially w.r.t. the size of the affected database. Indeed, such an algorithm computes consistent answers to the query by one evaluation of the query per repair, but in general the number of repairs may be exponential in the number of constraint violations, respectively in the size of the affected database. Actually, this is the best one may asymptotically expect to achieve for general inconsistent databases and universal constraints, unless P = NP, given that consistent query answering is Π_2^P-hard in such a setting for nonrecursive Datalog$^\neg$ queries (cf. Chomicki [2007] and references therein).

Hence, it is particularly relevant to assess whether some smarter strategies can be conceived for special settings, in order to have an algorithm that both implements localized repair computation and linearly scales w.r.t. the size of the database. More precisely, the aim is at localization strategies that, for the recombination step, exploit situations where the number of repairs which need to be considered for consistent query answering is linear in the size of the affected database. In the next subsection we formally elaborate such a strategy based on repair factorization, whereas a logic-programming based implementation is described in Section 6.

5.2 Repair Factorization

In this section, we present a technique that factorizes repairs into independent components. The basic idea is to partition the affected part $A = D \cap E$ of the database D w.r.t. a repair envelope E into disjoint subparts $A_1, \ldots, A_m$, such that the repairs of A are obtained by combining the repairs of $A_1, \ldots, A_m$ in all possible ways. Given a repair envelope E for D and χ, a partitioning $E_1, \ldots, E_m$ of E is a *factorization* of E for D and χ, if

$$rep(D) = \{(D \setminus E) \cup R_1 \cup \cdots \cup R_m \mid R_i \in rep(D \cap E_i), 1 \leq i \leq m\}. \qquad (7)$$

Towards sufficient conditions for factorization, we define a repair-compliant partitioning as follows.

Definition 5.1. Let E be a repair envelope for a database D for a schema $\chi = \langle \Psi, \Sigma \rangle$. A partitioning $E_1, \ldots, E_m$ of E is *repair-compliant*, if (1) it is *constraint-bounded*, that is, constraint-bounded facts from E belong to the same component E_i, and (2) for all $R \in rep_\chi(D \cap E)$ and $R_i \in rep_\chi(D \cap E_i)$, $1 \leq i \leq m$, $R \setminus E = R_i \setminus E_i$.

Example 5.1. Consider a schema with relations $r(A, B)$ and $s(B, C)$ which have the keys A and B, respectively. Let $D = \{r(a_1, b_1), r(a_1, b_2), r(a_2, b_1), r(a_2, b_3), r(a_3, b_1), s(b_1, c_1), s(b_1, c_2), s(b_3, c_3)\}$. Its conflict set is $C = D \setminus \{r(a_3, b_1), s(b_3, c_3)\}$, which is a repair envelope. Note that the safe part of D is $S = \{r(a_3, b_1), s(b_3, c_3)\}$. The partitioning $C_{r_1} = \{r(a_1, b_1), r(a_1, b_2)\}$, $C_{r_2} = \{r(a_2, b_1), r(a_2, b_3)\}$

and $C_s = \{s(b_1, c_1), s(b_1, c_2)\}$ of C is repair-compliant: It is easily verified that it is constraint-bounded (constraint-bounded facts are exactly the pairs of facts in each partition). Moreover for each repair $R \in \mathit{rep}(D \cap E)$ we have $R \setminus E = \emptyset$, since it consists in dropping one fact of each conflicting pair. Therefore, the same is true for each partition, i.e., $R \setminus C_{r_1} = \emptyset$ for $R \in \mathit{rep}(D \cap C_{r_1})$, $R \setminus C_{r_2} = \emptyset$ for $R \in \mathit{rep}(D \cap C_{r_2})$, and $R \setminus C_s = \emptyset$ for $R \in \mathit{rep}(D \cap C_s)$.

By means of a repair-compliant partitioning, we can factorize the repair of $A = D \cap E$ into the repair of the (mutually disjoint) parts $A_i = A \cap E_i = D \cap E_i$ of A, for $i = 1, \ldots, m$. The repairs for each A_i are confined to $F \cup E_i$ for a fixed set of facts F, and by the abstract properties (SIP), (DPE), and (DIS) of the preference ordering, they can be easily combined with the repairs for all other parts A_j, as shown next.

THEOREM 5.2 (FACTORIZATION). *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, and let E be a repair envelope for D. Then, every repair-compliant partitioning $E_1, \ldots, E_m$ of E is a factorization of E for D and χ.*

PROOF. We need to show that $\mathit{rep}(D) = \{(D \setminus E) \cup R_1 \cup \cdots \cup R_m \mid R_i \in \mathit{rep}(D \cap E_i), 1 \leq i \leq m\}$. Since E is a repair envelope for D and χ, we know that $\mathit{rep}(D) = \{(D \setminus E) \cup R \mid R \in \mathit{rep}(D \cap E)\}$. Hence, it is sufficient to prove that:

($\subseteq$) $R \in \mathit{rep}(D \cap E)$ implies $R = R_1 \cup \ldots \cup R_m$ and $R_i \in \mathit{rep}(D \cap E_i)$ for $1 \leq i \leq m$;
($\supseteq$) every $R \in \{R_1 \cup \ldots \cup R_m \mid R_i \in \mathit{rep}(D \cap E_i), 1 \leq i \leq m\}$ is a repair of $D \cap E$.

($\subseteq$) Let $R \in \mathit{rep}(D \cap E)$. Then, by repair-compliance of $E_1, \ldots E_m$, $R = F \cup R_E$, where $F = R \setminus E$ and $R_E \subseteq E$. Consider $R_i = F \cup (R_E \cap E_i)$ for $1 \leq i \leq m$. It remains to show that $R_i \in \mathit{rep}(D \cap E_i)$ for $1 \leq i \leq m$. Towards a contradiction first assume that $R_i \not\models \Sigma$ for some $1 \leq i \leq m$. Then, there exists some $\sigma \in \mathit{ground}(\Sigma)$ such that $R_i \not\models \sigma$. Thus, $R_i \models \mathit{body}(\sigma)$, which implies $R \models \mathit{body}(\sigma)$, and $R_i \not\models \mathit{head}(\sigma)$. However, $R \models \mathit{head}(\sigma)$ must hold since $R \models \Sigma$ by hypothesis. This means that there exists a head atom $B(\vec{y})$ of σ which is true in R. Since $R_i \not\models \mathit{head}(\sigma)$, none of the built-in predicates of σ is true and therefore $B(\vec{y})$ is a fact such that $B(\vec{y}) \in E_j$, $j \neq i$. Since the partitioning is constraint-bounded, it follows that $\mathit{body}(\sigma) \subseteq F$ and $\mathit{head}(\sigma) \cap E_i = \emptyset$. Thus no repair of the form $F \cup R'_{E_i}$ of $D \cap E_i$ such that $R'_{E_i} \subseteq E_i$ can exist, a contradiction to the repair compliance of $E_1, \ldots, E_m$. This proves $R_i \models \Sigma$ for every $i = 1, \ldots, m$.

Consequently, $R_i \in \mathit{rep}(D \cap E_i)$ iff there exists no $R'_i \in \mathit{rep}(D \cap E_i)$ such that $R'_i <_{D \cap E_i} R_i$ and $R'_i \models \Sigma$. Assume such an R'_i exists. Then $R'_i = F \cup R'_{E_i}$ and thus by (DIS) $R'_{E_i} <_{D \cap E_i} R_{E_i}$. By (DPE) we would conclude for $R'_E = (R \cap E_1) \cup \ldots \cup R'_{E_i} \cup \ldots \cup (R \cap E_m)$, that $R'_E <_{D \cap E} R_E$, which implies $R' <_{D \cap E} R$ for $R' = F \cup R'_E$. Furthermore, $R' \models \Sigma$. (Otherwise there exists some $\sigma \in \mathit{ground}(\Sigma)$ such that $R' \models \mathit{body}(\sigma)$ and $R' \not\models \mathit{head}(\sigma)$, while $R \models \sigma$. We can conclude that $\mathit{body}(\sigma) \subseteq R'_i$, and since $R'_i \models \Sigma$ we obtain $R' \models \mathit{head}(\sigma)$, a contradiction). Together with $R' <_{D \cap E} R$, however, this contradicts $R \in \mathit{rep}(D \cap E)$. Hence, $R_i \in \mathit{rep}(D \cap E_i)$, for $1 \leq i \leq m$.

($\supseteq$) Let $R \in \{R_1 \cup \ldots \cup R_m \mid R_i \in \mathit{rep}(D \cap E_i), 1 \leq i \leq m\}$. We show that $R \in \mathit{rep}(D \cap E)$. Towards a contradiction suppose $R \not\models \Sigma$, that is, $R \not\models \sigma$

for some $\sigma \in ground(\Sigma)$. By definition of a repair-compliant partitioning, we conclude that $R = F \cup (R_i \cap E_i) \ldots \cup (R_m \cap E_m)$, where $F = R_i \setminus E_i$ for any $1 \leq i \leq m$. Consequently, $R \models body(\sigma)$ implies $R_i \models body(\sigma)$ for some $1 \leq i \leq m$ by constraint-boundedness. However, $R_i \not\models head(\sigma)$ (otherwise $R \models head(\sigma)$), which contradicts $R_i \in rep(D \cap E_i)$. Hence, $R \models \Sigma$.

It remains to show that there is no $R' \in rep(D \cap E)$ such that $R' <_{D \cap E} R$. Assume the contrary and let $F = R' \setminus E$. Then by (DIS) (disjunctive split), either (i) $R' \cap E_i <_{D \cap E_i} R \cap E_i$ or (ii) $R' \setminus E_i <_{(D \cap E) \setminus E_i} R \setminus E_i$ holds for each $i = 1, \ldots, m$. Case (i) leads to a contradiction with $R_i \in rep(D \cap E_i)$, however, since it implies $F \cup (R' \cap E_i) <_{D \cap E_i} R_i$ and $F \cup (R' \cap E_i) \models \Sigma$ (otherwise $R' \not\models \Sigma$). So (ii) must hold for every $i = 1, \ldots, m$. As shown by the recursive argument below, it follows that $R' \setminus E <_{(D \cap E) \setminus E} R \setminus E$, which however, by repair-compliance of $E_1, \ldots, E_m$, is equivalent to $F <_{\emptyset} F$, a contradiction. To see this, note that we can apply (DIS) to $R' \setminus E_i <_{(D \cap E) \setminus E_i} R \setminus E_i$ w.r.t. E_j for any $1 \leq j \neq i \leq m$, and arrive in a similar situation as above: either (i') $(R' \setminus E_i) \cap E_j <_{((D \cap E) \setminus E_i) \cap E_j} (R \setminus E_i) \cap E_j = R' \cap E_j <_{D \cap E_j} R \cap E_j$, or (ii') $(R' \setminus E_i) \setminus E_j <_{((D \cap E) \setminus E_i) \setminus E_j} (R \setminus E_i) \setminus E_j$. Now (i') leads to a contradiction as in (i), and therefore (ii') must hold. Iterating this argument $m - 1$ times yields $R' \setminus (E_1 \cup \ldots \cup E_m) <_{(D \cap E) \setminus (E_1 \cup \ldots \cup E_m)} R \setminus (E_1 \cup \ldots \cup E_m)$, which is equivalent $R' \setminus E <_{(D \cap E) \setminus E} R \setminus E$. This proves $R \in rep(D \cap E)$. □

Note that Condition (2) of Definition 5.1 is trivially satisfied for $\mathbf{C}_0$ constraints. Furthermore, it is immaterial for $\mathbf{C}_1$ constraints under the standard envelope $E = C^*$.

PROPOSITION 5.3. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, and let E be a repair envelope for D. If either (1) $\Sigma \subseteq \mathbf{C}_1$ and $E = C^*$ or (2) $\Sigma \subseteq \mathbf{C}_0$, then every constraint-bounded partitioning $E_1, \ldots, E_m$ of E is repair-compliant.*

Thus, for the practically important classes of constraints $\mathbf{C}_1$ and $\mathbf{C}_0$, repair-compliant partitionings, and thus factorizations, can be obtained by a constraint-bounded partitioning of C^*, respectively by a constraint-bounded partitioning of any repair envelope. Consequently, for $\mathbf{C}_0$ and the canonical envelope $E = C$, Equation (7) can be rewritten to:

$$rep(D) = \{(D \setminus C) \cup R_1 \cup \cdots \cup R_m \mid R_i \in rep(C_i), 1 \leq i \leq m\}, \tag{8}$$

where $C_1, \ldots, C_m$ is a constraint-bounded partition of C.

Example 5.2. Let χ consist of the relation $p(x, y, z)$ and the functional dependency $f : p(x, y, z) \wedge p(x, y', z') \supset z = z'$ (which is of class $\mathbf{C}_0$), and consider the database $D = \{p(a_i, b_j, c_k) \mid 1 \leq i \leq m \wedge 1 \leq j, k \leq \ell\}$. The conflict set C consists of all tuples in D, since each pair of facts of the form $p(a_i, b_j, c_k)$ and $p(a_i, b_{j'}, c_{k'})$ with $k \neq k'$ witnesses a violation of f. The partitioning $C_1, \ldots, C_m$ of C, where $C_i = \{p(a_i, b_j, c_k) \in C\}$, $1 \leq i \leq m$, is constraint-bounded and thus, by Proposition 5.3, repair-compliant, and by Theorem 5.2, a factorization. Every C_i has ℓ repairs, while D has ℓ^m repairs in total. In particular, the repairs of D are of the form $R_1 \cup \cdots \cup R_m$, where each R_i is a repair for C_i, according to Equation (8).

We finally remark that under particular preference relations, Condition (2) for repair-compliance (see Definition 5.1) might be relaxed. For instance, in case of the prototypical preorder $\leq_D$, that is, set inclusion w.r.t. symmetric difference, it is sufficient that the repairs of $D \cap E_i$ coincide outside E_i on a fixed part: for all $1 \leq i, j \leq m$, $R_i \in rep(D \cap E_i)$ and $R_j \in rep(D \cap E_j)$ implies $R_i \setminus E_i = R_j \setminus E_j$.

Furthermore, we note that we can compute efficiently repair-compliant partitionings of arbitrary repair envelopes for $\mathbf{C}_0$ constraints and of the standard envelope $E = C^*$ for $\mathbf{C}_1$ constraints, for instance, using techniques for computing the connected components of a graph. Note that each E_i is a union of connected components of the graph with nodes in E and edges between each pair of constraint-bounded facts. In this respect, point out that techniques exploiting different graph (and hypergraph) representations of conflicts in data have been introduced and used in Arenas et al. [2001] and Chomicki and Marcinkowski [2005].

5.2.1 *Recombination of Independent Factors.* We are now in the position to show how the notion of factorization can be used to optimize query answering from inconsistent databases. To this end, we proceed in two directions:

—First, for a user query Q, we investigate when some of the components of a factorization $E_1, \ldots, E_m$ do not entail repairs in all possible ways to answer Q. Intuitively, this happens when a single repair of the affected part $D \cap E_i$ is sufficient for answering Q.

—Second, we investigate how to improve the naive usage of Equation (7), by discussing scenarios where consistent answers to Q can be obtained by independently processing the different components, rather than combining their repairs in all possible ways.

We focus here on nonrecursive Datalog queries $Q = \langle q, \mathcal{P} \rangle$. Since they can be effectively unfolded to a union of conjunctive queries with a single head predicate q, we assume that queries are already in this form, i.e., $\mathcal{P} = \{\rho_1, \ldots, \rho_n\}$, where $head(\rho_j) = q(\vec{t}_j)$. We denote by $n(Q)$ the number of conjunctive queries, i.e., rules in $\mathcal{P}$ and by $v(Q)$ the maximal number of variables appearing in any ρ_j.

In order to understand whether for the component E_i not all repairs are needed, we consider a *rewriting* of Q w.r.t. E_i. It aims at determining whether a particular repair for E_i is sufficient to answer Q. Roughly, for each ρ_j in $\mathcal{P}$ a test rule $\rho_{i,j}$ is created, which for each repair R for E_i yields a cautious overestimate of the result of ρ_j over the safe database plus R; the overestimates for all ρ_j are then collected into an overestimate of the result of Q. If over all repairs R for E_i a single minimal overestimate exists, then we can simply use the corresponding R in computing the consistent answer of Q w.r.t. D.

To define $\rho_{i,j}$, we introduce three subsets of $body(\rho_j)$: a set $\beta_{i,j}^{in}$ of atoms that may change value in different repairs, a set $\beta_{i,j}^{out}$ of non-built-in atoms unaffected by all repairs, and a set $\beta_{i,j}^{bin}$ of built-in atoms that are connected to atoms in these two sets. In detail,

$$\begin{aligned}
\beta^{in}_{i,j} &= \{p(\vec{x}) \in body(\rho_j) \mid \exists\theta : p(\vec{x}\theta) \in E_i\},\\
\beta^{out}_{i,j} &= \{p(\vec{x}) \in body(\rho_j) \mid \forall\theta : p(\vec{x}\theta) \notin E\},\\
\beta^{bin}_{i,j} &= \{\phi(\vec{x}) \in body(\rho_j) \mid \forall x \in \vec{x}\exists p(\vec{x}') \in \beta^{in}_{i,j} \cup \beta^{out}_{i,j} \ \wedge x \in \vec{x}'\},
\end{aligned}$$

where θ ranges over ground substitutions, $p \in \Psi$ (i.e., non-built-in), and ϕ denotes a built-in predicate; note that $\beta^{out}_{i,j} = \beta^{out}_{i',j}$ for all E_i and $E_{i'}$. In words, for each ρ_j in $\mathcal{P}$, $\beta^{in}_{i,j}$ are the body atoms in ρ_j that match with some fact in E_i; $\beta^{out}_{i,j}$ are the body atoms of ρ_j that cannot be matched with any fact in the envelope E via ground variable substitution; and $\beta^{bin}_{i,j}$ are the built-in (body) atoms in ρ_j which join some atom in $\beta^{out}_{i,j}$ or $\beta^{in}_{i,j}$.[2]

Example 5.3. Consider a schema χ with a ternary relation r_1, two binary relations r_2 and r_3, and a unary relation r_4, where the first argument for each relation is the key. Assume also that relations r_1 and r_2 are disjoint, i.e., it holds that $r_2(x, y) \wedge r_3(x, y') \supset y \neq y'$. Let $D = \{r_1(a, b, c), r_1(a, b, d), r_2(c, b), r_3(c, b), r_4(b)\}$. Its conflict set is $C = D \setminus \{r_4(b)\}$, which is a repair envelope. The partitioning $E_1 = \{r_1(a, b, c), r_1(a, b, d)\}$, $E_2 = \{r_2(c, b), r_3(c, b)\}$ of C is repair-compliant (it is indeed constraint-bounded), and therefore is a factorization of E for D and χ. Consider the query $Q = \langle q, \mathcal{P}\rangle$, where $\mathcal{P} = \{q(x) \leftarrow r_1(x, y, z), r_2(w, y), q(x) \leftarrow r_1(x, y, z), r_3(w, y), r_4(y)\}$, i.e., such that $n(Q) = 2$ and $v(Q) = 4$. Then, we easily obtain $\beta^{in}_{1,1} = \beta^{in}_{1,2} = \{r_1(x, y, z)\}$, $\beta^{in}_{2,1} = \{r_2(w, y)\}$, $\beta^{in}_{2,2} = \{r_3(w, y)\}$, $\beta^{out}_{1,1} = \beta^{out}_{2,1} = \emptyset$, $\beta^{out}_{1,2} = \beta^{out}_{2,2} = \{r_4(y)\}$, $\beta^{bin}_{1,1} = \beta^{bin}_{1,2} = \beta^{bin}_{2,1} = \beta^{bin}_{2,2} = \emptyset$.

Armed with the above notions, we now define the rewriting Q_i of Q w.r.t. E_i. The body of the test rule $\rho_{i,j}$ consists simply of all atoms in $\beta^{in}_{i,j}$, $\beta^{out}_{i,j}$, and $\beta^{bin}_{i,j}$. Its head contains all head variables of ρ_j that occur in $\beta^{in}_{i,j}$, plus further join variables from atoms in $\beta^{in}_{i,j}$ to atoms in the body of ρ_j outside $\beta^{in}_{i,j} \cup \beta^{out}_{i,j}$; intuitively, the join variables provide additional context information. The head of $\rho_{i,j}$ also contains an identifier d_j that marks the contribution of $\rho_{i,j}$ to the result of Q_i; finally, since technically each head $\rho_{i,j}$ must have the same predicate, but the resulting lists of head variables for $\rho_{i,j}$ may have different lengths, we pad all lists to the same length using d_j. In detail,

Definition 5.2. Let $E_1, \ldots, E_m$ be a factorization of a repair envelope E for a database D, let Q be a nonrecursive (unfolded) Datalog query as above, and let $d_1, \ldots, d_{n(Q)} \notin \mathcal{U}$ be fresh constants. We define Q_i, the rewriting of Q w.r.t. E_i, as follows:

$$Q_i = \langle q_i, \mathcal{P}_i\rangle, \quad \mathcal{P}_i = \{\rho_{i,j} \mid 1 \le j \le n(Q)\},$$

where

(1) $head(\rho_{i,j}) = q_i(d_j, \vec{u}_{i,j}, \vec{v}_{i,j}, \vec{d}_j)$ and $body(\rho_{i,j}) = \beta^{in}_{i,j} \cup \beta^{out}_{i,j} \cup \beta^{bin}_{i,j}$ (=: $\beta_{i,j}$);

(2) the arity of q_i is $v(Q) + 1$;

[2] Note that we consider safe queries. We could keep all built-in predicates if we allowed for unsafe rules or if we added respective domain predicates, ranging over the active domain, to make the rule safe.

(3) $\vec{u}_{i,j}$ are the variables from $\vec{t}_j$ (in any order), that is, the variables in $head(\rho_j) = q(\vec{t}_j)$, which occur in some $p(\vec{x}) \in \beta^{in}_{i,j}$;

(4) $\vec{v}_{i,j}$ are the variables (in any order) not occurring in $\vec{u}_{i,j}$ but in some $p(\vec{x}) \in \beta^{in}_{i,j}$ and some $p'(\vec{y}) \in body(\rho_j) \setminus \beta^{in}_{i,j}$, unless $p'(\vec{y}) \in \beta^{out}_{i,j}$ or p' is built-in, and all variables from $\vec{y}$ occur in $\beta^{in}_{i,j}$;

(5) $\vec{d}_j = d_j, \ldots, d_j$ is a padding to the arity of q_i;

Note that in Condition (2), we use $v(Q)+1$ for simplicity (smaller values are possible). In Condition (4), the join variables from $\beta^{in}_{i,j}$ to atoms outside $\beta^{in}_{i,j} \cup \beta^{out}_{i,j}$ are collected in $\vec{v}_j$; as an optimization, built-in atoms having all their variables in $\beta^{in}_{i,j}$ can be omitted.

Example 5.4. In the setting of Example 5.3, we have $Q_1 = \langle q_1, \{\rho_{1,1}, \rho_{1,2}\}\rangle$, with $\rho_{1,1} = q_1(d_1, x, y, d_1, d_1) \leftarrow r_1(x, y, z)$, and $\rho_{1,2} = q_1(d_2, x, y, d_2, d_2) \leftarrow r_1(x, y, z), r_4(y)$, and $Q_2 = \langle q_2, \{\rho_{2,1}, \rho_{2,2}\}\rangle$, with $\rho_{2,1} = q_2(d_1, y, d_1, d_1, d_1) \leftarrow r_2(w, y)$, and $\rho_{2,2} = q_2(d_2, y, d_2, d_2, d_2) \leftarrow r_3(w, y), r_4(y)$.

Example 5.5. Continuing Example 5.1, the partitioning C_{r_1}, C_{r_2}, C_s of C is repair-compliant, and thus by Theorem 5.2 a factorization. The query $Q = \langle q, \{q(x) \leftarrow r(x, y), s(y, z).\}\rangle$ has $n(Q) = 1$ and $v(Q) = 3$. For C_{r_1} and C_{r_2}, we obtain $\beta^{in}_{r_1,1} = \beta^{in}_{r_2,1} = \{r(x, y)\}$, and for C_s, $\beta^{in}_{s,1} = \{s(y, z)\}$. Furthermore, $\beta^{out}_{r_1,1} = \beta^{out}_{r_2,1} = \beta^{out}_{s,1} = \emptyset$ and $\beta^{bin}_{r_1,1} = \beta^{bin}_{r_2,1} = \beta^{bin}_{s,1} = \emptyset$. We thus have $Q_{r_1} = \langle q_{r_1}, \{q_{r_1}(d_1, x, y, d_1) \leftarrow r(x, y)\}\rangle$ and $Q_{r_2} = \langle q_{r_2}, \{q_{r_2}(d_1, x, y, d_1) \leftarrow r(x, y)\}\rangle$, while $Q_s = \langle q_s, \{q_s(d_1, y, d_1, d_1) \leftarrow s(y, z)\}\rangle$.

Recall that the rewriting Q_i should allow to identify components for which a single repair is sufficient to evaluate the original query Q. As we show now, this is the case if considering all possible repairs R for a component, Q_i has a single minimal (w.r.t. set inclusion) result over the safe database plus R. We call such components *singular*.

Definition 5.3. Let $E_1 \ldots, E_m \subseteq E$ be a factorization of a repair envelope E for a database D. Let Q be a nonrecursive (unfolded) Datalog query, and let $Q_i = \langle q_i, \mathcal{P}_i\rangle$ be a rewriting of Q w.r.t. E_i, $1 \leq i \leq m$. Denote by $amin(Q_i)$ the set of the evaluations $Q_i[(D \setminus E) \cup R]$, where $R \in rep(D \cap E_i)$, which are minimal w.r.t. set inclusion. We then call E_i *singular*, if $R \setminus E_i = R' \setminus E_i$ for all $R, R' \in rep(D \cap E_i)$ and $|amin(Q_i)| = 1$.

Example 5.6. For the setting of Example 5.4, we have $D \setminus E = \{r_4(b)\}$, and $rep(D \cap E_1) = \{R_1, R_2\}$, with $R_1 = \{r_1(a, b, c)\}$, and $R_2 = \{r_1(a, b, d)\}$, $rep(D \cap E_2) = \{R_3, R_4\}$, with $R_3 = \{r_2(c, b)\}$ and $R_4 = \{r_3(c, b)\}$. It is easy to see that $Q_1[D \setminus E \cup R_1] = Q_1[D \setminus E \cup R_2] = \{(d_1, a, b, d_1, d_1), (d_2, a, b, d_2, d_2)\}$, and therefore $|amin(Q_1)| = 1$, i.e., E_1 is singular. Furthermore, $Q_2[D \setminus E \cup R_3] = \{(d_1, b, d_1, d_1, d_1)\}$, $Q_2[D \setminus E \cup R_4] = \{(d_2, b, d_2, d_2, d_2)\}$. Hence $|amin(Q_1)| \neq 1$, i.e., E_2 is not singular.

Example 5.7. In our Example 5.5, $D \cap C_{r_1}$ has the two repairs $\{r(a_1, b_1)\}$ and $\{r(a_1, b_2)\}$, and $amin(Q_{r_1}) = \{\{(d_1, a_1, b_1, d_1), (d_1, a_3, b_1, d_1)\}, \{(d_1, a_1, b_2, d_1), (d_1, a_3, b_1, d_1)\}\}$. Similarly, $D \cap C_{r_2}$ has two repairs, $\{r(a_2, b_1)\}$ and $\{r(a_2, b_3)\}$, and $amin(Q_{r_2}) = \{\{(d_1, a_2, b_1, d_1), (d_1, a_3, b_1, d_1)\},$

$\{(d_1, a_2, b_3, d_1), (d_1, a_3, b_1, d_1)\}\}$. Finally, $D \cap C_s$ has the two repairs $\{s(b_1, c_1)\}$ and $\{s(b_1, c_2)\}$, but $amin(Q_s)$ is the singleton $\{\{(d_1, b_1, d_1, d_1), (d_1, b_3, d_1, d_1)\}\}$, i.e., C_s is a singular component.

For a singular component E_i it is sufficient to consider a repair R that yields the minimal evaluation w.r.t. Q_i, that is, such that $Q_i[(D \setminus E) \cup R] \in amin(Q_i)$. Hence, we can pick one such repair for each singular component and add it to the safe database, obtaining a set of facts that has not to be altered and can safely be combined with repairs of other components without altering the consistent answers of Q. We call such a set a *query-safe part*.

Definition 5.4. Let $E_1, \ldots, E_m$ be a factorization of a repair envelope E for a database D, where $E_1, \ldots, E_\ell$ are singular components, and let Q be a nonrecursive Datalog query. We call $S_Q = (D \setminus E) \cup R_1 \cup \cdots \cup R_\ell$ a *query-safe part* of D w.r.t. Q, if each R_i, $1 \leq i \leq \ell$, is an arbitrary repair of $D \cap E_i$, such that $Q_i[(D \setminus E) \cup R_i] \in amin(Q_i)$.

In Example 5.6, E_1 is the only singular component, and $R_1 = \{r_1(a, b, c)\}$ is one such repair of $D \cap E_1$. Hence, $S_Q = \{r_1(a, b, c), r_4(b)\}$ is a query-safe part of D. Similarly in Example 5.7, C_s is a singular component, and $S_Q = \{r(a_3, b_1), s(b_3, c_3), s(b_1, c_2)\}$ is a query-safe part. We then have the following result.

PROPOSITION 5.4. *Let $E_1, \ldots, E_m$ be a factorization of a repair envelope E for a database D, let Q be a nonrecursive Datalog query, and let $S_Q = (D \setminus E) \cup R_1 \cup \cdots \cup R_\ell$ be a* query-safe part *of D w.r.t. Q. Then,*

$$ans_c(Q, D) = \bigcap_{R_{\ell+1} \in rep(D \cap E_{\ell+1})} \cdots \bigcap_{R_m \in rep(D \cap E_m)} Q[S_Q \cup R_{\ell+1} \cdots \cup R_m]. \qquad (9)$$

If all components E_i are singular ($\ell = m$), query answering can be carried out by considering an arbitrary query-safe part of D. In this ideal case, the cost for query answering amounts to checking for all $1 \leq i \leq m$ that $|amin(Q_i)| = 1$ and that $R \setminus E_i = R' \setminus E_i$ for all $R, R' \in rep(D \cap E_i)$ (recall that this is trivial for $\mathbf{C}_0$ constraints and the standard envelope C^* in case of $\mathbf{C}_1$ constraints) as well as eventually computing $Q[S_Q]$. Note that checking for singular components E_i and determining a repair R_i, such that $Q_i[(D \setminus E) \cup R_i] \in amin(Q_i)$, can be carried out by processing the components independently of each other and is polynomial if the local repairs can be computed in polynomial time. For each component E_i, the effort generally depends on the number of its repairs.

Moreover, *irrelevant* components can be easily detected by syntactic checks: If $\beta^{in}_{i,j} = \emptyset$ for $1 \leq j \leq v(Q)$, then trivially $|amin(Q_i)| = 1$. In this case, if $R \subseteq (D \setminus E) \cup E_i$ for all $R \in rep(D \cap E_i)$, we can even tolerate inconsistency, that is, we do not need to repair the component for consistent query answering and can skip it in the query-safe part S_Q.

Example 5.8. Concluding Example 5.6, we apply Equation (9) as follows:

$$ans_c(Q, D) = \bigcap_{R \in rep(D \cap E_2)} Q[S_Q \cup R] = \bigcap_{R \in \{\{r_2(c,b)\}, \{r_3(c,b)\}\}} Q[\{r_1(a, b, c), r_4(b)\} \cup R],$$

and we get $ans_c(Q, D) = \{(b)\}$; this is the correct result. We can proceed analogously also for Example 5.7, where there are two nonsingular components.

However, we will show below that in this case a further optimization is possible.

Even if nonsingular components are present, it may be possible to "parallelize" consistent query answering without a need for recombination. A simple case is if there is just a single atom in the body of the query such that for a set of components this is the only atom that unifies with facts from these components. Then, the query can be independently evaluated over these components since there is no "interference" between the components w.r.t. the query evaluation, like a join in the query that unifies with facts belonging to different components in this set. We call such a set of components *decomposable*.

Definition 5.5. Let $E_1, \ldots, E_m$ be a factorization of a repair envelope E for D. A set of components $E_1, \ldots, E_\ell$ is *decomposable* w.r.t. query Q, if they satisfy:

(1) $\beta^{in}_{i,k} = \beta^{in}_{j,k}$, for every $1 \leq i, j \leq \ell$ and $1 \leq k \leq n(Q)$;
(2) $|\beta^{in}_{i,k}| = 1$, for every $1 \leq i \leq \ell$ and $1 \leq k \leq n(Q)$;
(3) $R_i \setminus E_i = R_j \setminus E_j$, for every $R_i \in rep(D \cap E_i)$, $R_j \in rep(D \cap E_j)$, $1 \leq i, j \leq \ell$.

Query answering over decomposable components can be parallelized, and can be combined with singular components as follows.

THEOREM 5.5. *Let $E_1, \ldots, E_m$ be a factorization of a repair envelope E for a database D, and let Q be a nonrecursive Datalog query. Suppose that $E_1, \ldots, E_\ell$ are singular components, with query safe part S_Q, and that $E_{\ell+1}, \ldots E_k$ is a set of decomposable components w.r.t. query Q. Then*

$$ans_c(Q, D) = \bigcap_{R_{k+1} \in rep(D \cap E_{k+1})} \cdots \bigcap_{R_m \in rep(D \cap E_m)} \bigcup_{i=\ell+1}^{k} \left(\bigcap_{R_i \in rep(D \cap E_i)} Q[S_Q \cup R_i \cup R_{k+1} \cup \cdots \cup R_m] \right). \quad (10)$$

PROOF. We first show that

$$Q[X \cup R_{\ell+1} \cup \cdots \cup R_k] = \textstyle\bigcup_{i=\ell+1}^{k} Q[X \cup R_i], \quad (11)$$

for every $R_j \in rep(D \cap E_j)$, $\ell < j \leq k$, and for every set of facts X. We have $Q = \langle q, \mathcal{P} \rangle$, where $\mathcal{P} = \{\rho_1, \ldots, \rho_n\}$ is a set of "*not*"-free rules ρ_j. Consider any ground instance ρ'_j of ρ_j, and an atom $p(\bar{t})$ occurring in the body of ρ'_j that is satisfied by $X \cup R_{\ell+1} \cup \cdots \cup R_k$. Then either $p(\bar{t}) \in X$ or $p(\bar{t}) \in R_h$, for some $\ell < h \leq k$. Furthermore, since $E_{\ell+1}, \ldots, E_k$ are decomposable w.r.t. Q, in the case where $p(\bar{t}) \in R_h \setminus (X \cup \bigcap_{i=\ell+1}^{k} R_i)$, there is no atom $p'(\bar{t}')$ in the body of ρ'_j which belongs to $R_{h'} \setminus R_h$, for some $\ell < h' \neq h \leq k$. Indeed, by Condition (3), $R'_h \setminus R_h = R_{h'} \cap E_{h'}$, and thus $p'(\bar{t}') \in E_{h'}$ would hold, while $p(\bar{t}) \in E_h$ holds. Conditions (1) and (2) would imply that $p(\bar{t})$ and $p'(\bar{t}')$ are instances of the same atom in the body of ρ_j, and thus $p(\bar{t}) = p'(\bar{t}')$. This contradicts $E_h \cap E_{h'} = \emptyset$.

As a consequence, the body of ρ'_j is satisfied by $X \cup R_{\ell+1} \cdots \cup R_k$ iff it is satisfied by $X \cup R_h$, for some $\ell + 1 \leq h \leq k$. Since ρ_j is nondisjunctive and “*not*”-free, it follows that $Q^{(j)}[X \cup R_{\ell+1} \cdots \cup R_k] = \bigcup_{i=\ell+1}^{k} Q^{(j)}[X \cup R_i]$, where $Q^{(j)} = \langle q, \{\rho_j\}\rangle$, and that

$$\begin{aligned} Q[X \cup R_{\ell+1} \cdots \cup R_k] &= \bigcup_{j=1}^{n} Q^{(j)}[X \cup R_{\ell+1} \cdots \cup R_k] = \bigcup_{j=1}^{n} \bigcup_{i=\ell+1}^{k} Q^{(j)}[X \cup R_i] \\ &= \bigcup_{i=\ell+1}^{k} \bigcup_{j=1}^{n} Q^{(j)}[X \cup R_i] = Q[X \cup R_{\ell+1}] \cup \cdots \cup Q[X \cup R_k]. \end{aligned}$$

This proves (11). To conclude the proof, by Proposition 5.4 and setting $X = S_Q \cup R_{k+1} \cup \cdots \cup R_m$ the consistent answers to Q w.r.t. D are:

$$ans_c(Q, D) = \bigcap_{R_{\ell+1} \in rep(D \cap E_{\ell+1})} \cdots \bigcap_{R_m \in rep(D \cap E_m)} Q[S_Q \cup R_{\ell+1} \cdots \cup R_m]$$

By Equation (11), we then get:

$$ans_c(Q, D) = \bigcap_{R_{\ell+1} \in rep(D \cap E_{\ell+1})} \cdots \bigcap_{R_m \in rep(D \cap E_m)} \big(Q[X \cup R_{\ell+1}] \cup \cdots \cup Q[X \cup R_k]\big),$$

from which the result follows by Boolean algebra (recall that for any sets A, $B_1, \ldots, B_k$, it holds that $\bigcap_{B \in \{B_1,\ldots,B_k\}}(A \cup B) = A \cup \bigcap_{B \in \{B_1,\ldots,B_k\}} B$). □

By this result, we can take any (but not necessarily all) singular components, any decomposable components, and parallelize query answering. As mentioned above, singular components can always be identified efficiently given their local repairs. Furthermore, also a maximal set of decomposable components can be identified efficiently given the local repairs. In fact, all maximal such sets—which are pairwise disjoint—can be determined efficiently (note that the relation $E_{i_1} \equiv_Q E_{i_2}$ iff the set E_{i_1}, E_{i_2} is decomposable w.r.t. Q, is an equivalence relation). In particular, if the factorization stems from a repair-compliant partitioning, then Condition (3) is always fulfilled, and the effort depends only on the check for (1) and (2). Roughly, in total the computational effort by exploiting singular and decomposable components reduces from the global combination of all local repairs of the components to taking the union of the results of local consistent query answering.

Example 5.9. In Example 5.7, C_{r_1} and C_{r_2} are nonsingular components. Since $n(Q) = 1$ and for both, C_{r_1} and C_{r_2}, we have $\beta^{in}_{r_i,1} = \{r(x, y)\}$, $1 \leq i \leq 2$, the set $\{C_{r_1}, C_{r_2}\}$ is decomposable w.r.t. Q. By Theorem 5.5, the query $Q = \langle q, \{q(x) \leftarrow r(x, y), s(y, z).\}\rangle$ can be evaluated independently over C_{r_1} and C_{r_2}, taking, e.g., the query-safe part $S_Q = \{r(a_3, b_1), s(b_3, c_3), s(b_1, c_2)\}$, into account. Specifically, for C_{r_1}, we must compute $Q[S_Q \cup \{r(a_1, b_1)\}] \cap Q[S_Q \cup \{r(a_1, b_2)\}]$, which yields $\{(a_3)\}$. For C_{r_2}, we must compute $Q[S_Q \cup \{r(a_2, b_1)\}] \cap Q[S_Q \cup \{r(a_2, b_3)\}]$, which yields $\{(a_3), (a_2)\}$. Therefore, $ans_c(Q, D) = \{(a_3), (a_2)\}$. As can be checked, this is the correct result.

Importantly, by virtue of Proposition 5.4 and Theorem 5.5, one can consistently answer queries in polynomial time which do not belong to any class shown to be tractable in the literature [Chomicki et al. [2004b, 2004a]; Fuxman et al.

[2005]; Grieco et al. [2005]]. We exemplify this for the scenario discussed in the following Example 5.10. The query there contains a nonkey to nonkey join, that is, a join between nonkey positions, and hence is not in any class shown to be tractable (see also Section 7). Nonetheless, we will show how to compute consistent answers for this query in polynomial time. We illustrate this in two steps: First, we exemplify this by exploiting the techniques from above for *restricted inputs*; it would not be clear how to do this from previous results either (which hinge on conditions on the query and the constraints, but are insensitive to the actual data). Second, we combine the factorization techniques with a simple grounding strategy, which then yields a polynomial evaluation method for the query over *arbitrary inputs*. We finally extend this approach to a class of queries (of slightly different form) for which consistent query answering is tractable.

Example 5.10. Inspired by a typical information system supporting university administration, we consider a schema with the relations $student(IDS, First, Last, Address)$, $prof(IDP, First, Last)$, and $dean(IDD, First, Last)$, where IDS, IDP, and IDD are the respective keys. As a further constraint, we have the inclusion dependency $dean(x, y, z) \supset prof(x, y, z)$.

Suppose we want to know the identifiers of professors having their last name in common with a student. The query $Q = \langle q, \{q(x_2) \leftarrow student(x_1, y_1, z, w_1), prof(x_2, y_2, z).\}\rangle$ extracts them; note that it involves a nonkey to nonkey join. Now let D be a database with the conflict set $C = \{student(0815, johann, meier, addr1), student(0815, hans, meier, addr1), prof(4711, markus, schmidt), prof(4711, mark, schmidt), dean(1111, egbertus, neumann), prof(1111, egbertus, neumann))\}$. C is a repair envelope, which can be readily factorized into three components, two of them containing a pair of tuples from E with equal key value over *prof* and *student*, respectively, and one containing the remaining facts of C. As easily verified, all these components are singular. (This would not hold for the canonical repair envelope C^*.)

If D' is a database yielding the conflict set $C' = \{student(0815, johann, meier, addr1), student(0815, johann, maier, addr1), student(4711, bodo, schmied, addr2), student(4711, bodo, schmid, addr2)\}$, and professors called *meier*, *maier*, *schmied*, and *schmid*, respectively, exist. Then, obviously $E' = C'$ is a repair envelope, and we can, for example, again factorize into components containing tuples with equal key values (two this time). In this case the respective components are not singular, but decomposable.

Hence, we observe that as long as violations are restricted to only one of the relations *student* or *prof*, we end up with singular and decomposable components such that Theorem 5.5 remains effective with a single set of decomposable components. The same is true for database instances where violations affect both relations, but where all violations w.r.t. one of the relations end up in singular components.

We also remark that, as long as violations are restricted to only one of the relations or all components are singular, computing consistent answers to the query in Example 5.10 is feasible in polynomial time, even if we had a further relation *staff* of the same structure as *prof* in the schema and the query extended to a union of conjunctive queries by adding the rule $q(x_2) \leftarrow$ $\mathit{student}(x_1, y_1, z, w_1), \mathit{staff}(x_2, y_2, z)$. As well, we could add the exclusion dependency $\mathit{dean}(x_1, y_1, z_1) \wedge \mathit{staff}(x_2, y_2, z_2) \supset x_1 \neq x_2 \vee y_1 \neq y_2 \vee z_1 \neq z_2$ to the scenario and still would obtain tractability for consistently answering the query. Observe that in all these scenarios, in order to obtain a repair envelope, we do not need to take the entire conflict closure into account. Rather we can restrict the envelope to constraint bounded facts over tuples of constants occurring in the conflict set (or simply to the conflict set, as done in Example 5.10). Furthermore, the resulting envelope can trivially be factorized into components with identical key values.

For arbitrary violations however, conflicts may interfere w.r.t. the query. In our student and professor example, too many nonsingular and nondecomposable components might emerge, such that Theorem 5.5 can not be applied to establish tractability of the query, even if local repairs are computable in polynomial time. Nevertheless, tractability of this query can be established by means of a refined factorization strategy, as shown next.

5.2.2 *Factorization and Query Grounding.* Our basic factorization approach might be combined with other techniques in order to get further tractability results. One such technique is to reduce general nonrecursive queries $Q = \langle q, \mathcal{P} \rangle$, where q has arity k, to ground (Boolean) queries by means of unification: for a tuple $\vec{c} = (c_1, \ldots, c_k)$ of constants, unify each rule $\rho \in \mathcal{P}$ with q in the head with $q(\vec{c})$, that is, apply to ρ a substitution θ of constants to ρ's head variables such that $\mathit{head}(\rho\theta) = q(\vec{c})$; if no such θ exists (e.g., for $q(b, X) \leftarrow p(X, c)$ and $q(a, b)$), simply delete ρ. Denote the resulting query by $Q_{\vec{c}}$;[3] then

$$\mathit{ans}_c(Q, D) = \{\vec{c} = (c_1, \ldots, c_k) \mid c_1, \ldots, c_k \text{ occur in } D \text{ or } Q,\ \mathit{ans}_c(Q_{\vec{c}}, D) \neq \emptyset\}. \tag{12}$$

That is, we can parallelize query answering for each tuple.

We can view $Q_{\vec{c}}$ as a refinement of Q, which is equivalently obtained by adding equality literals $x = c_i$ in the rule bodies in Q. As for consistent query answering, this does not affect decomposability of components E_i (since this property does not depend on built-ins), and preserves their singularity, that is, whenever E_i is singular w.r.t. Q, then E_i is also singular w.r.t. $Q_{\vec{c}}$. Thus, the number of singular components can only increase, and in benign cases only few components that are nonsingular nor decomposable remain. In these cases, we may exploit Equation (12) and Theorem 5.5 to compute consistent query answers in polynomial time.

In the light of this important observation, we review the scenario in Example 5.10.

Example 5.11. Reconsider the schema and $Q = \langle q, \{q(x_2) \leftarrow \mathit{student}(x_1, y_1, z, w_1), \mathit{prof}(x_2, y_2, z).\}\rangle$ in Example 5.10. Assume that we have arbitrary violations in both the relations *student* and *prof*. Consider for the tuple $\vec{c} = c_1$ the query $Q_{\vec{c}} = \langle q, \{q(c_1) \leftarrow \mathit{student}(x_1, y_1, z, w_1), \mathit{prof}(c_1, y_2, z).\}\rangle$. Then, all

[3] Notice that if $n = 0$ (i.e., $\vec{c}$ is void), $\mathit{ans}_c(Q, D) \neq \emptyset$ means that Q is consistently true, false otherwise.

components in the discussed factorization for *prof*, except at most one, are singular w.r.t. $Q_{\vec{c}}$, and all components for *student* are singular or (jointly) decomposable. Indeed, the only component for *prof* which may not be singular is the one involving tuples with key c_1 (if such a component exists). Using Theorem 5.5, we can evaluate $Q_{\vec{c}}$ in polynomial time w.r.t. the database D. By evaluating $Q_{\vec{c}}$ for each constant c_1 occurring in D, we thus can compute the consistent query answer of Q w.r.t. D as in Equation (12) in polynomial time. Hence, the query Q is tractable.

This example illustrates that queries of the form of—and in a constraint setting as in—our student and professor example are polynomial in data complexity for arbitrary inputs. This will be formally established by Proposition 5.6.

As a further optimization, we note here that it is possible to do even better than evaluating query $Q_{\vec{c}}$, for each possible tuple $\vec{c} = (c_1, \ldots, c_k)$ in the output. In fact, rather than fixing the output of Q at all positions i to c_i, we may fix them one by one. After each step, we test whether sufficiently many components are singular w.r.t. the modified query Q' (i.e., only few components are neither decomposable nor singular, which means that Q' can be polynomially evaluated); if not, then we fix the next position. This is repeated until the answer is yes or no position remains.

The resulting query Q' covers all output tuples for Q that have at the respective positions the values fixed in Q'. Similarly, we construct a modified query Q'' for a possible tuple $\vec{c}'$ for Q that is not covered by Q'; by repeating this process, we will collect a list of queries $Q', Q'', \ldots$ that we can evaluate using Theorem 5.5, such that the union of all their answers will give us $ans_c(Q, D)$. Noticeably, the singularity tests for the modified queries can be done without evaluating their rewritings as in Definition 5.2. A detailed study and refinement of this strategy remains for future work.

We conclude this section by noting that the tractability result in Example 5.10 can be generalized, and identify an entire class of queries for which we can establish that consistent query answering is polynomial in data complexity without looking at the inconsistencies in the actual database; that is, we single out a purely "syntactic" condition for tractability.

Let $\mathcal{Q}_{1k\exists}$ denote the class of all conjunctive queries Q (without built-in literals) over schemas that have at most one key constraint per relation, such that except in at most one atom, key positions are always head variables or constants in Q. Note that the query in Example 5.10 satisfies this condition.

PROPOSITION 5.6. *For $\mathcal{Q}_{1k\exists}$, consistent query answering is polynomial in data complexity.*

Note that $\mathcal{Q}_{1k\exists}$ does not fall into any tractable query class in the literature (cf. Section 7).[4] The class may be further generalized, e.g., by allowing between

[4] A dichotomy result in Fuxman and Miller [2007] would suggest that some queries in this class are co-NP-complete (e.g., queries with only nonkey to nonkey joins). However, the proof there assumes that queries have at least one variable in the key positions of each atom, which is not the case for the considered queries.

relations r and s, where r occurs in the query and s does not, limited exclusion dependencies and inclusion dependencies of the form $r(\vec{x}) \supset s(\vec{x})$ or $s(\vec{x}) \supset r(\vec{x})$ under further syntactic restrictions.

6. LOGIC PROGRAMMING FOR CONSISTENT QUERY ANSWERING

According to several proposals in the literature, consistent answers from inconsistent databases can be computed by encoding the constraints in the schema by means of a Datalog program using unstratified negation or disjunction, in such a way that the stable models of this program map to the repairs of the database. A framework that abstracts from several logic programming formalizations in the literature (such as Greco et al. [2003]; Arenas et al. [2003]; Barceló and Bertossi [2003]) is introduced next.[5]

Definition 5.6. Let $Q = \langle q, \mathcal{P} \rangle$ be a nonrecursive Datalog$^{\neg}$ query over $\chi = \langle \Psi, \Sigma \rangle$. A *logic specification for querying* χ *with* Q is a (safe) Datalog$^{\vee,\neg}$ program $\Pi_\chi(Q) = \Pi_\Sigma \cup \Pi_Q$ such that, for a given $D \in \mathcal{D}(\chi)$,

(1) $rep_\chi(D) \rightleftharpoons \mathrm{SM}(\Pi_\Sigma \cup D)$, and
(2) $ans_c(Q, D) = Q'[D]$, where $Q' = \langle q, \Pi_\chi(Q) \rangle$, that is, $ans_c(Q, D) = \{\vec{t} \mid q(\vec{t}\,) \in M$ for each $M \in \mathrm{SM}((\Pi_\Sigma \cup \Pi_Q) \cup D)\}$, where Π_Q is a nonrecursive safe Datalog$^{\neg}$ program,

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

In the above definition, Π_Σ is that portion of $\Pi_\chi(Q)$ that encodes the integrity constraints in Σ, whereas Π_Q represents an encoding of the logic program $\mathcal{P}$ in the user query Q (examples of instantiations of the above logic framework are given in Appendix E).

Encoding repair computation by means of logic programs 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. In fact, extensions to the Datalog$^{\vee,\neg}$ language that allow, for instance, to handle priorities and weight constraints [Leone et al. 2006; Simons et al. 2002], 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 (as in Arenas et al. [2003]).

However, with current (yet still improving) implementations of stable model engines, such as DLV [Leone et al. 2006] or Smodels [Simons et al. 2002], query evaluation over large data sets quickly becomes infeasible because of lacking scalability. The source of complexity in evaluating the program $\Pi_\chi(Q)$ lies in the

[5] Other logic formalizations proposed in the data integration setting [Lembo et al. 2002; Bertossi et al. 2002; Calì et al. 2003b; Bravo and Bertossi 2003] also fit in our framework, provided that the *retrieved global database* [Lenzerini 2002] is computed. Notice also that other logic-based approaches to data integration, based on abductive logic programming [Arieli et al. 2004] and ID-logic [Nuffelen et al. 2004], do not fit this framework.

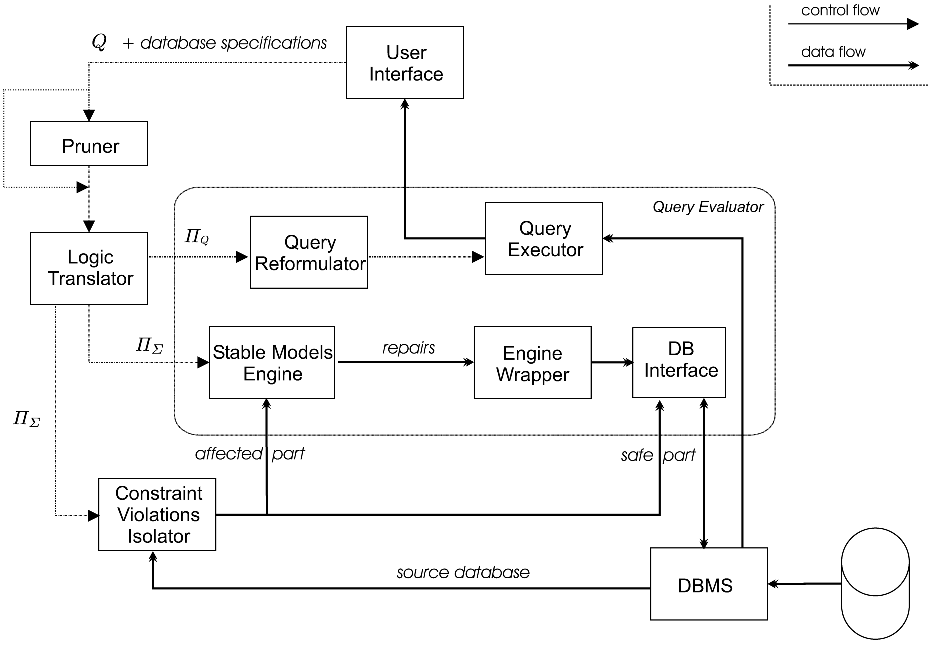


Fig. 3. System architecture.

conflict resolution module Π_Σ. Indeed, while Π_Q, which is in general a nonrecursive Datalog$^\neg$ program, can be evaluated in polynomial time with respect to underlying databases (data complexity) [Dantsin et al. 2001], Π_Σ is in general a Datalog$^{\vee,\neg}$ program [Greco et al. 2003], whose evaluation data complexity is at the second level of the polynomial hierarchy [Dantsin et al. 2001].

6.1 General Architecture for Repair Compilation

The localization properties discussed in Section 4 and Section 5 may be used to optimize consistent query answering from inconsistent databases. Indeed, computing the repairs for D may be done in practice by evaluating the program Π_Σ only over the affected part of the database D, rather than on the whole D as obtained by a straight evaluation of the program $\Pi_\chi(Q)$ over D (Item 1 in Definition 6.1). We thus propose an approach to optimize query answering that implements the strategies in Equation (6) and Equation (10). In practice, we just need an architecture in which a stable model engine used to retrieve one repair at a time is interfaced with a DBMS that evaluates the query over the repair augmented with the safe part of D. Figure 3 shows a concrete architecture, whose components have the following functionalities:

—*Pruner:* It takes the user query Q and the schema χ, and produces an equivalent specification (w.r.t. Q), stripping off relations and constraints irrelevant for answering Q. This is a preprocessing step, which is not discussed in detail here.

—*Logic Translator:* It takes the specification of χ relevant for Q returned by the Pruner, and produces the logic program $\Pi_\chi(Q) = \Pi_\Sigma \cup \Pi_Q$, according to some encoding proposed in the literature. In our tests, we used the mapping in Calì et al. [2003b] and Grieco et al. [2005].

—*Constraint Violations Isolator:* It is responsible of processing the program Π_Σ to produce a set of SQL views isolating the safe and the affected parts of the database at hand. When strategies in Section 5.2 are to be applied (cf. Equation (10)), the affected part is in fact provided in terms of a factorization.

—*Stable Models Engine:* It takes as input the affected database (or, in fact, each of the components involved in a factorization, when recombination is going to be implemented via Equation (10)) and computes the repairs using the program Π_Σ. In our implementation, we used the DLV system [Leone et al. 2006].

—*Engine Wrapper:* It wraps the output of the Stable Models Engine, by asking the engine for one repair at time. In our implementation, this is done with the JAVA Wrapper module available for DLV.[6] If the constraints are not in the class $\mathbf{C}_1$, it also has to filter from any repair the facts that are not in the envelope E—see condition (ii) of Proposition 5.1.

—*DB Interface:* It does the interfacing between the Stable Models Engine and the DBMS, in which it stores the repair computed by the Stable Model Engine—in fact, the safe part is not modified in this process. After a new repair is stored, it notifies the Query Executor.

—*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 implementing the recombination step. As discussed in Section 5, based on the query Q and the database D, with safe part S, it may choose to apply either the repair factorization strategy in Equation (10), possibly with some further optimizations as the one discussed in Section 5.2.2, or the basic approach in Equation (6). As for the strategy in Equation (6), the module stores in the DBMS the result of the execution in a table. When the first repair of the affected part (intersected with E), say R_1, is processed, the table is initialized with the result of the evaluation of Q over $R_1 \cup S$. Then, for each other repair R_i, the table is updated by filtering those tuples that do not occur in the answer to Q over $R_i \cup$. After the last repair is computed, the table is returned to the user. A similar strategy is applied for Equation (10), with the major difference that now the process is repeated for each component involved in the factorization rather than once for the whole affected part. Eventually, to recombine the results of the evaluation over each component (as to implemented Equation (10)), some further temporary tables are used in the DBMS.

Note that in the case where D is consistent, query processing resorts to standard query evaluation over the DBMS, with some overhead for checking constraint violations by the *Constraint Violations Isolator*. In fact, in this case

[6] http://www.mat.unical.it/wrapper/index.html

$player_m{}^{\mathcal{M}_{D_0}}$:

10	*Totti*	*RM*	$'11'$
9	*Ronaldinho*	*BC*	$'11'$

$team_m{}^{\mathcal{M}_{D_0}}$:

RM	*Roma*	10	$'10'$
BC	*Barcelona*	8	$'11'$
RM	*Real Madrid*	10	$'01'$

$coach_m{}^{\mathcal{M}_{D_0}}$:

7	*Capello*	*RM*	$'11'$

Fig. 4. The database of our running example after marking.

the *Query Executor* module evaluates the query directly over $S = D$, since no repair is produced by the *Stable Models Engine*.

6.2 Grouped Repair Computation

As discussed above, the *Query Executor* module implements the recombination step by executing some SQL statements, for *each* repair computed by the *Stable Models Engine*. As a further optimization we next consider the idea of grouping these repairs in such a way that a single SQL statement may be evaluated over more than one repair at time. This can be done using a marking strategy, and independently on whether recombination is implemented by means of the basic approach in Section 5.1 or by repair factorization. Indeed, in the former case, we may think of grouping all the repairs of the affected part, while in the latter case, for each component E_i involved in the factorization, we may think of grouping all the repairs of $D \cap E_i$ (see Equation (10)).

Let $R_1, \ldots, R_n$ be the repairs which we want to simultaneously process on the DBMS, indexed using the order in which the *Stable Models Engine* computes them. In each relation s, we add an auxiliary attribute *mark*, leading to a new relation s_m. The values for *mark* are strings of bits 0, 1. To each fact $s(\vec{t}) \in D$, we associate a mark $b =' b_1 \ldots b_n'$ such that, for every $1 \leq i \leq n$, $b_i = 1$ if $s(\vec{t})$ belongs to R_i, and $b_i = 0$ otherwise. The marked tuple $\vec{t}, b$ is stored in the corresponding relation s_m. The extensions of all s_m constitute the *marked* database, denoted by D_m. Note that the facts in the safe database can be marked without preprocessing: their mark is $'11 \ldots 1'$, since they belong to every repair R_i. In our running example, the marked database derived from the repairs in Figure 1 is shown in Figure 4. In a first approximation, the marked database may be considered as having its tables altered with an extra column which stores the mark.

A nonrecursive Datalog$^\neg$ query $Q = \langle q, \mathcal{P} \rangle$ is reformulated into an SQL query over D_m by first normalizing the rules in $\mathcal{P}$ and then converting each rule r into a separate SQL query SQL_r. Let $r : h(\vec{x}_0) \leftarrow B(\vec{x})$ be a safe rule of form

$$p_0(\vec{x}_0) \leftarrow p_1(\vec{x}_1), \ldots, p_l(\vec{x}_l),\ not\ p_{l+1}(\vec{x}_{l+1}), \ldots,\ not\ p_{l+k}(\vec{x}_{l+k}),[7] \qquad (13)$$

and 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 $1 \leq j \leq k_i$. Then, 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,

[7] For the sake of simplicity and w.l.o.g. we assume that variables occurring in $\vec{x}_0$ are all distinct.

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 $0 \leq i' \leq l$.)

Informally, SQL_r selects tuples for the head predicate of r, thereby respecting not only the join conditions given by the body of r, but also the marks of the joined tuples. Marks corresponding to negative literals are inverted and missing tuples (which belong to no repair) are viewed as marked by $'0 \ldots 0'$ (see Appendix D for details).

Eventually, 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:

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

where SUMBIT denotes an aggregate function that, given m marks (i.e., bit strings), returns the mark given by bitwise OR. By such a view for the query predicate q, denoted q_m, the consistent answers to the query Q are obtained through the statement SQL_Q:

```
SELECT $a_1, \ldots a_n$ FROM $q_m$ WHERE $mark =' 1 \ldots 1'$.
```

It computes the consistent query answers by selecting the facts that are true in all repairs.

Example 6.1. The query in our running example has two rules: $r_1 : q(x) \leftarrow player(x, y, z)$ and $r_2 : q(x) \leftarrow team(v, w, x)$. Their normalized versions are:

$$\begin{array}{l} r_1' : q(y_{0,1}) \leftarrow player(y_{1,1}, y_{1,2}, y_{1,3}), y_{0,1} = y_{1,1}; \\ r_2' : q(y_{0,1}) \leftarrow team(y_{1,1}, y_{1,2}, y_{1,3}), y_{0,1} = y_{1,3}. \end{array}$$

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 $mark =' 11'$;
```

SQL_Q yields on D_m the tuples (*8*), (*9*), and (*10*); they are the consistent answers to Q.

The query SQL_Q has the following property (the proof is given in Appendix D).

PROPOSITION 6.1. *Let D be a database for $\chi = \langle \Psi, \Sigma \rangle$, let Q be a nonrecursive Datalog$^\neg$ query over it, and let $R_1, ..., R_n$ be databases such that $R_i = R'_i \cap E$, where E is a weak repair envelope for D and R'_i is a repair for $A = D \cap E$. Then, SQL_Q computes on D_m the set of tuples $\bigcap_{i=1}^{n}\{\vec{t} \mid \vec{t} \in Q[R_i \cup S]\}$, for $S = D \setminus E$.*

Note that when $R'_1, \ldots, R'_n$ are all repairs for A, then the tuples computed by SQL_Q are the consistent answers to Q w.r.t. D—see, again, Equation (6).

A limitation to the scalability of the marking strategy is that all safe tuples must be marked with $'11\ldots1'$, since they belong to each repair. However, we can avoid this and evaluate a reformulated query on a database instance in which only affected tuples have been marked. In more detail, with each 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 part of r^D, respectively. Also, we construct the database instance A' by replacing each relation symbol r in A_D with r_{aff}, and the database instance S' by replacing each relation symbol r in S_D with r_{safe}, i.e., we have that ${r_{aff}}^{A'} = \{\vec{t} \mid r(\vec{t}\,) \in A\}$ and ${r_{safe}}^{S'} = \{\vec{t} \mid r(\vec{t}\,) \in S\}$. Then, given a query $Q = \langle q, \mathcal{P} \rangle$, where $\mathcal{P}$ is assumed to be normalized, over a schema $\chi = \langle \Psi, \Sigma \rangle$, we proceed as follows:

—for each rule $r : h(\vec{x}_0) \leftarrow B(\vec{x})$ of form (13) 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 relation $p_j \in \Psi$ by the literals *not* $p_{aff_j}(\vec{x_j})$, *not* $p_{safe_j}(\vec{x}_j)$; let $B'_i(\vec{x})$ be the result;

—we replace r with the rules $r_i : h(\vec{x}_0) \leftarrow B'_i(\vec{x})$, for $1 \leq i \leq n$;

—in the SQL statement SQL_{r_i} for r_i, we replace every $p_{safe_{j\,m}}$ by p_{safe_j}, and $p_{safe_j}.mark$ by $'1\ldots1'$.

One can show that the SQL reformulation of the query Q as described above, denoted SQL'_Q, yields over the partially marked database $S' \cup A'_m$ the same result as SQL_Q over the fully marked database D_m. That is, for the reformulation SQL'_Q only the affected tuples have to be marked. Notice that SQL'_Q 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 A but not on the size of S. For example, for 10 constraint violations involving two tuples each, the required marking space is $2{*}10{*}2^{10}$ bits = 2.5 KB, independently of the size of 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.

Further optimizations concerning the marking strategy may be carried out, in particular DBMS dependent techniques can be deployed, but are beyond the scope of this paper.

Here we conclude by noticing that throughout this section, we have fixed the length of the markings to coincide with the total number n of repairs of the affected part. Thus, according to Proposition 6.1, the DBMS can be queried just once for recombining the results of the repairs with the safe part and for getting consistent answers. However, the length n of the marks can be fixed independently of the actual number of repairs. Indeed, if $R_1, \ldots, R_m$ are the repairs of the affected part and if $n < m$, then applying Proposition 6.1 $\lceil m/n \rceil$ times (once for each group of n repairs as they are incrementally computed by the *Stable Models Engine*) and intersecting the partial results is sufficient. In the extreme case where $n = 1$, this approach would amount to standard evaluation without markings. Section 8 extensively discusses benefits of this grouping strategy and suggests appropriate values for the length of marking strings.

7. OTHER APPROACHES TO CONSISTENT QUERY ANSWERING

Efficient computation of consistent answers to queries on inconsistent databases has received increasing attention recently [Arenas et al. 1999; Fuxman and Miller 2007; Grieco et al. 2005; Chomicki and Marcinkowski 2005; Chomicki et al. 2004a]. The quoted works single out settings in which this task is feasible with polynomial data complexity, by imposing suitable restrictions on both the form of the constraints in the database schema and on the query language. Briefly, they differ from our work as follows.

—Other papers, Arenas et al. [1999], Fuxman and Miller [2007], and Grieco et al. [2005], considered only repairs according to the prototypical repair ordering $\leq_D$ introduced in Arenas et al. [1999]. In contrast, our results cover a generic range of repair orderings, and may be extended to repair semantics based on preference orderings violating the properties in Section 3.1; for example, Chomicki and Marcinkowski [2005] 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 4.2, Lemma 4.3, and Theorem 4.4 can be established similarly.

—Next, all papers above present methods for consistent query answering that hinge on the query and the constraints, but not on the actual database. Arenas et al. [1999]; Fuxman and Miller [2007]; Grieco et al. [2005] employ *first-order query rewriting* as the main approach, where the user query q is rewritten into a new query q' expressed in first-order logic, such that the evaluation of q' over every underlying databases D returns the consistent answers to q w.r.t. D. Chomicki et al. [2005, 2004a] use a constraint violation hypergraph to reason efficiently about consistent query answers, which also works on an arbitrary database D. In contrast, our localization approach looks at the actual database D and exploits the structure of the data and possible repairs. In case of benign violations, consistent query answering can be done efficiently using our approach, also when the first-order rewriting and the hypergraph approach as presented are not applicable.

—Furthermore, our techniques can handle very general and powerful forms of integrity constraints and/or queries that do not fall within the settings studied previously in the literature, which usually make quite limiting assumptions.

In the following we briefly comment on the above mentioned papers, starting from those based on first-order rewriting. The first results in this direction were given by Arenas et al. [Arenas et al. 1999]. The method proposed in that paper was proved to be sound and complete for queries expressed in quantifier-free first-order logic without disjunction, in the presence of binary universal integrity constraints (a limited fragment of our constraint classes $\mathbf{C}_1$ and $\mathbf{C}_2$). Celle at al. [2000] proposed an extension of the above technique that applies to a slightly more general class of (still binary universal) integrity constraints, and described an implementation of their algorithm. However, the setting considered in that work is still very restrictive.

More recently, Fuxman and Miller [2007] singled out a class of first-order rewritable queries, called $\mathcal{C}_{forest}$, for database schemas containing only key constraints (at most one for each relational predicate). Roughly speaking, $\mathcal{C}_{forest}$ contains conjunctive queries for which joins involving nonkey positions must satisfy a particular acyclicity condition. Furthermore, the queries must not contain self-joins, that is, repeated relation symbols, nor nonfull nonkey-to-key joins, that is, joins between relation symbols r and s that involve a nonkey position in r and a strict subset of the key positions of s (and vice versa). The given query rewriting algorithm has been adapted in the ConQuer system [Fuxman et al. 2005] to deal directly with Select-Project-Join queries expressed in SQL; moreover, also aggregate expressions were allowed.

Grieco et al. [2005] extended the class $\mathcal{C}_{forest}$ by allowing some forms of nonfull nonkey-to-key joins and also considered additional exclusion dependencies in the database schema. They gave sufficient conditions and algorithms for consistent query answering via first-order rewriting for this setting. To ensure rewritability, only limited interaction between the query atoms and every exclusion dependency between relation symbols r and s is allowed: r and s cannot both occur in the query; if r or s occurs in the query, then the dependency must involve subsets of the keys of r and s; other exclusion dependencies between r and q and between s and p, respectively, where p and q are (not necessarily distinct) relation symbols occurring in the query, are forbidden.

As mentioned above, the approach of Chomicki et al. [2004a, 2005] to consistent query answering is not based on first-order query rewriting, but constructs a hypergraph that represents the conflicts in the database, and exploits this conflict hypergraph to reason about the consistent query answers by considering independent sets. The technique enables consistent query answering in polynomial time (in data complexity) for specific combinations of denial constraints and queries: projection-free queries in relational algebra (each variable is an output-variable) in presence of arbitrary denial constraints [Chomicki et al. 2004a], and closed simple conjunctive queries (where projections are allowed, but no joins) in presence of functional dependencies over different relations [Chomicki and Marcinkowski 2005]. For the latter queries, all components of the natural factorization of the standard repair envelope w.r.t. any database are clearly singular. Hence, Proposition 5.4 reestablishes that this class of queries can be handled in polynomial time. The approach of Chomicki et al. is implemented in the Hippo System [Chomicki et al. 2004a, 2004b].

These results are interesting from a practical point of view, and are often supported by experimental validations on large amounts of inconsistent data [Fuxman et al. 2005; Chomicki et al. 2004a; Grieco et al. 2005]. However, they apply to quite narrow settings, with a particular repair semantics and very limited sets of constraints and/or queries.

Our optimization techniques, instead, cover a number of repair orderings from the literature, and address large classes of queries and constraints. Our repair factorization strategy can be applied, for instance, to a setting with generic unions of conjunctive queries on database schemas that contain generic key constraints and exclusion dependencies (but also inclusion dependencies falling in the class $\mathbf{C}_1$). It does not impose a priori restrictions on the structure of the schema and the query, but instead conditions on the interaction between the actual inconsistency in the database and the query. For sufficiently benign violation behavior, consistent query answering in polynomial time will be achieved; this may be good enough if nonbenign violations rarely occur. Guaranteed worst-case polynomial time behavior can be enforced by limiting, as in other approaches, the structure of the query and the constraints.

Finally, we point out that our optimization techniques may be also combined with the other approaches. For example, we can exploit factorization to single out cases that are not first-order rewritable in general, but are so in the light of the actual conflicts that we localize in the database. This approach seems to be promising, as it enables the exploitation of consolidated relational database technology also in some settings that cannot be handled by current first-order rewriting techniques.

8. EXPERIMENTAL RESULTS

In this section, we present experimental results for evaluating the effectiveness of our approach and, specifically, the benefits of the localization techniques discussed in the paper.

8.1 Benchmark Databases and Compared Methods

As already mentioned, Hippo [Chomicki et al. 2004a, 2004b] and ConQuer [Fuxman et al. 2005; Fuxman and Miller 2007] are two noticeable prototype systems for consistent query answering from inconsistent databases. These systems are tailored for specific settings where this task is tractable and manage very specific classes of queries and constraints. For this reason, their performances have been tested on ad-hoc created benchmark databases; Fuxman et al. [2005] mainly generated synthetic data for the TCP-H specifications over a schema containing only single keys on relation symbols, and used queries of the class $\mathcal{C}_{forest}$, encoded into SQL, with aggregate expressions. Chomicki et al. [2004a] considered project-free queries over ternary relations and functional dependencies of the form $p(x, y, z) \wedge p(x, y', z') \supset z = z'$ over each such relation.

To assess the effectiveness of our localization approach, we need some novel scenario as it is not directly comparable with Hippo and Conquer, whose incomparability similarly requested specific benchmarks and data. Indeed, our techniques are designed for and can be used also in settings more general than those addressed by those systems (see Section 7).

On the other hand, if we focus on the classes of queries for which Hippo and ConQuer have been designed, it will come as no surprise that our approach pays in efficiency for its generality and expressiveness. And, in fact, we envisage an integrated architecture that switches to these more specialized and efficient systems whenever the query and the constraints fall in one of the classes they are able to deal with. To test our framework and the factorization techniques discussed in Section 5, we thus proceed as follows:

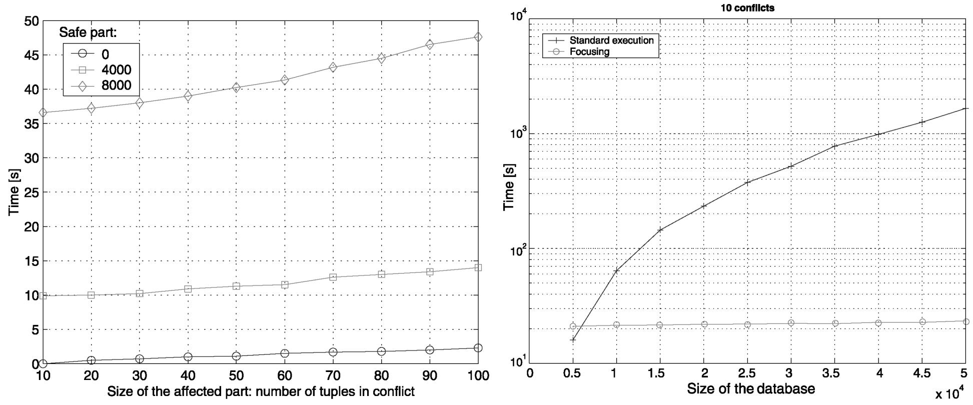
- —We first present experimental results for our running example (on football teams). The results show the advantages of focusing the computation by making use of the techniques discussed in Section 6.2, through the system described in Section 6.1, with respect to direct evaluating, through the DLV system, the logic specification for querying the inconsistent database with the query at hand.
- —We then focus on a test suite over the database schema χ_f^2 used in Chomicki et al. [2004a], which contains two ternary relations r_1 and r_2, and the functional dependencies $r_i(x, y, z) \wedge r_i(x, y', z') \supset z = z'$, with $i = 1, 2$, but we consider queries that involve projections, so that the system Hippo is not applicable. In this experiment, we show scalability of our approach with respect to a growing number of tuples in conflicts, and the advantage of combining repair factorization with markings.
- —We discuss the impact of the number of atoms in the query on the performance of the localization approach, by considering the schema χ_f^N which generalizes χ_f^2 with an increasing number of predicates.
- —Finally, we focus on a test suite that is based on the schema reported in Example 5.10, which refers to a typical information system supporting university administration. Experiments with this scenario aim to show the benefit of the factorization technique and the achievable scalability, in a context that is beyond the scope of applicability of other approaches in the literature focused on tractable settings of consistent query answering.

For the schemas, we generated random data following the idea of tuning the size of the safe part and the number of conflicts as in Chomicki et al. [2004a] and Fuxman et al. [2005].

All experiments have been carried out on a 1.6GHz Pentium IV with 512MB memory, by assessing the time needed for consistent query answering when the DLV system computes repairs of the affected part only, plus the time required for the recombination of the results in PostgreSQL. We always used a repair semantics based on the prototypical preorder $\leq_D$.

8.2 The Football Teams Example

We next discuss the performances of our approach and, specifically, its scaling w.r.t. the size of the safe database, on a simple scenario. For our running example, we built a synthetic data set D_{FT}, such that tuples in *coach* and *team* satisfy the key constraints issued on these relations, while tuples in *player* violate the



(a) (b)

Fig. 5. Football Team. (a) Execution time in DLV system w.r.t. size of the affected part. (b) Comparison with the optimization method.

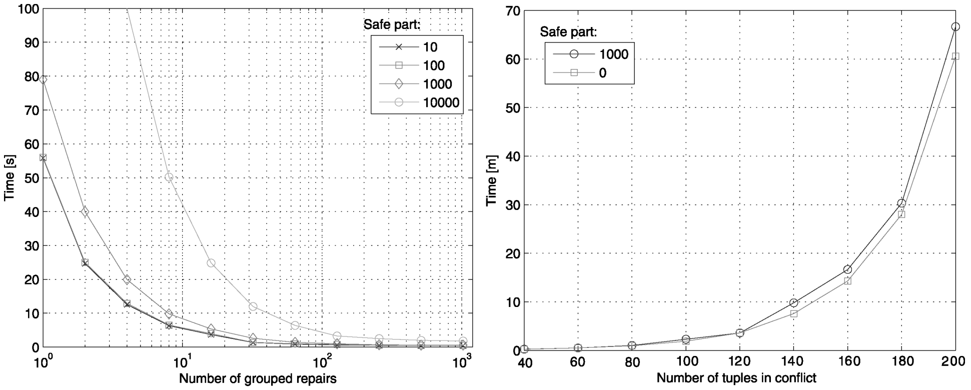
corresponding key constraint. Each violation consists of two facts that coincide on *Pcode* but differ on either *Pname* or *Pteam*; these facts constitute the affected part of D_{FT}. For our experiments, we consider the query $Q = \langle q, \mathcal{P} \rangle$ where $\mathcal{P} = \{q(x) \leftarrow player(x, y, z);\ q(x) \leftarrow team(v, w, x)\}$, and we encode our problem into a Datalog$^{\neg}$ program $\Pi_{\chi_0}(Q)$ in the line of [Calì et al. 2003b; Grieco et al. 2005] (the encoding used is the one given in Appendix E.1, in which we get rid of the encoding for the mapping). We first measure the execution time of the program $\Pi_{\chi_0}(Q)$ in DLV depending on the size of the affected part, while the size of the safe part is fixed to the values (i) 0, (ii) 4000, and (iii) 8000, respectively. We stress that values for the execution time of the DLV system refer to query answering with non-ground queries.

The results for this experiment, reported in Figure 5(a), show that the DLV system scales well w.r.t. the size of the affected part. Still the big size of the safe part appears to be the most limiting factor for an efficient implementation. Indeed, only 8000 facts (in absence of conflicting tuples) would require more than 35 seconds for consistent query answering.

The performance degradation under varying database size is further stressed in Figure 5(b), which shows a comparison (in log-scale) between consistent query answering using a single DLV program and the optimization approach proposed in this paper. As for the optimization approach, values on execution time include the cost of computing repairs of the affected database only, plus marking and evaluating the associated SQL query over marked relations. Specifically, we considered 10 violations and a marking string of 2^{10} bits, such that issuing one query over the database is sufficient to recombine the repairs of the affected part with the safe part. Interestingly, the growth of the running time of our optimization method under a varying database size is negligible.

8.3 Scalability Assessment

In the following experiments, we assessed the relevance of the strategy for grouping repair computation by focusing on the database χ_f^2. Indeed, so far,



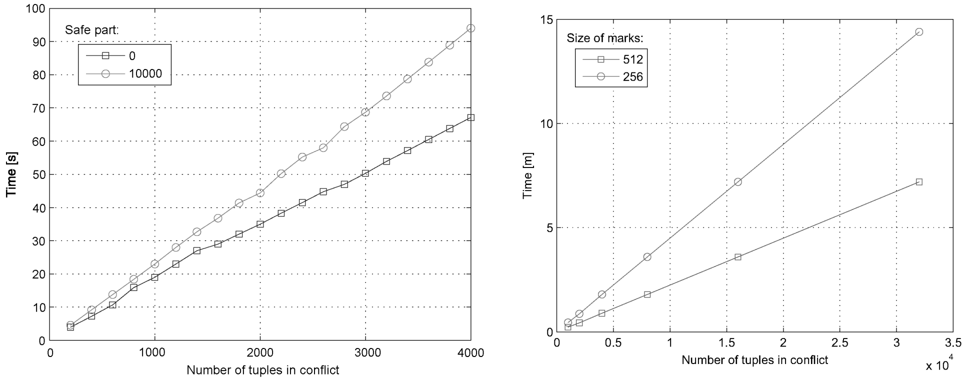
(a) (b)

Fig. 6. Query answering over χ^2_f. (a) Optimization method w.r.t. number n of grouped repairs, for a fixed number of conflicts. (b) Optimization method w.r.t. the size of the affected part, for $n = 2^9$.

we have assumed that the marking string is sufficient for storing all repairs for the affected part and, therefore, the DBMS has been queried just once for recombining the results of the localized repairs with the safe part only. But the reader may at this point wonder whether this approach is more efficient than processing each repair sequentially (one at a time).

Figure 6(a) answers the above question positively. It reports the time needed for answering the query $Q_f = \langle q_f, \mathcal{P}_f \rangle$ where $\mathcal{P}_f = \{q_f(y_1) \leftarrow r_1(x, y_1, z_1), r_2(x, y_2, z_2)\}$ w.r.t. the number n of repairs that are grouped and processed simultaneously on the DBMS. Specifically, we fixed 10 conflicts in the data (each involving two inconsistent tuples). Hence, for $n = 1$, we sequentially process each repair, while for $n = 2^{10}$, all the repairs are combined in the DBMS at the same time. The advantage of grouping repairs is evident, specifically by considering the scaling of the curves for different sizes of the safe part. Actually, we note that as the markings may grow exponentially with the size of the affected part, processing all the repairs at the same time is generally infeasible, since the length of the marking strings may exceed the maximum size allowed by the DBMS. For instance, with the bit string type of the PostgreSQL system, we may store marks up to $n = 2^{25}$ bits. In fact, to scale up to a few gigabytes we may resort to the *large objects* facilities of the system, or we may use well-known commercial DBMSs that provide embedded support for dealing with large binary objects. We point out that in our experiments we did not exploit such features, which instead may profitably be used within an engineered version of the prototype; indeed, in the following we used an incremental evaluation approach, eventually by grouping up to 1000 repairs at time—which is a value we experienced to be appropriate for the use with PostgreSQL.

In a second set of experiments over the query Q_f, we measured again the scalability w.r.t. the number of conflicts. In particular, we augmented the number of conflicts up to 100, and we fixed the marking string to 2^9 bits. The results shown in Figure 6(b) evidence the exponential scaling in the number of



(a) (b)

Fig. 7. Factorization strategy. Answering Q_f over χ_f^2 w.r.t. the size of the affected part: (a) Dependency on the safe part. (b) Dependency on the marking string for Q'_f.

conflicting tuples; this is indeed the best scaling we can expect for inconsistent databases in general, as the problem is co-NP-hard.

In fact, it is interesting to assess whether some nicer scaling can be obtained by applying the factorization strategy discussed in Section 5.2. In this respect, we notice that the given setting such that our factorization strategy can be applied. Indeed, the setting we are considering is basically the one described in Example 5.2, where each component contains only those facts witnessing a violation of the functional dependency over each of the two relations r_1 and r_2. Specifically, in our experiments, we fixed the structure of each component to contain 20 tuples and 1000 repairs (any pair of these tuples witnesses a violation of the dependency), and we generated some synthetic data for increasingly large number of independent components. In addition to the factorization strategy, we still exploit the grouping repair approach, by fixing the number of repairs simultaneously processed to 2^{10}.

The results obtained by applying the recombination strategy in Equation (10) are shown in Figure 7(a). Given the ability of independently processing the components, the scaling is now linear in the number of components and, hence, in the size of the whole affected part. In fact, query answering is feasible for a much larger number of constraint violations.

A similar experiment has been repeated for the query $Q'_f = \langle q'_f, \mathcal{P}'_f \rangle$ where $\mathcal{P}'_f = \{q'_f(y_1) \leftarrow r_1(x, y_1, z_1)\}$. After fixing the safe part to 10,000 tuples, we repeated the experiment up to a very large number of conflicts considering two different values for the parameter n which bounds the number of simultaneously processed repairs. Note in Figure 7(b) the linear scaling in the number of tuples in conflicts and the benefit from combining marking with the factorization strategy. In fact, this linear scaling is again due to the fact that components can independently be processed, so that the cost for query answering basically amounts to computing all repairs for each of these components and store them as marks into the database. As an extreme scenario, we note that the linear behavior of these latter tasks has been confirmed up to a million of tuples, for which the computation was about 9 hours.

Finally, the setting χ_f^2 has been generalized. We also considered the database χ_f^N and the query $Q = \langle q, \mathcal{P} \rangle$ where $\mathcal{P} = \{q(y_1) \leftarrow r_1(x, y_1, z_1), r_2(x, y_2, z_2), \ldots, r_N(x, y_N, z_N)\}$, for 10 constraint violations per relation and 2^{10} repairs simultaneously processed. In this scenario, we performed some experiments to assess the dependence of query answering on the number of atoms N in the query. The results are reported in Figure 8(a), which shows (as discussed in Section 6.2) an exponential dependency.

We conclude this overview on the behavior of the localization strategies presented in this article by considering the schema in Example 5.10. As a first case, we consider the query $Q' = \langle q, \{q(w_1) \leftarrow \mathit{student}(x_1, y_1, z, w_1), \mathit{prof}(x_2, y_2, z).\}\rangle$, which is a slight modification of the query Q in such example and which asks for addresses of those students that might possibly be involved in family relationships with professors.

Note that a full-inclusion dependency is issued over the schema (in addition to the keys), and that in Q' there is a join involving only nonkey positions, and also projection. Computing the consistent answers to such a query is co-NP-hard; indeed, the reader may check that Q' can be used to encode the MONOTONE-3SAT problem with minor modifications over the construction presented in [Fuxman and Miller 2007] (which is originally from Chomicki and Marcinkowski [2005]) to prove hardness of consistent query answering for queries over two distinct body literals.

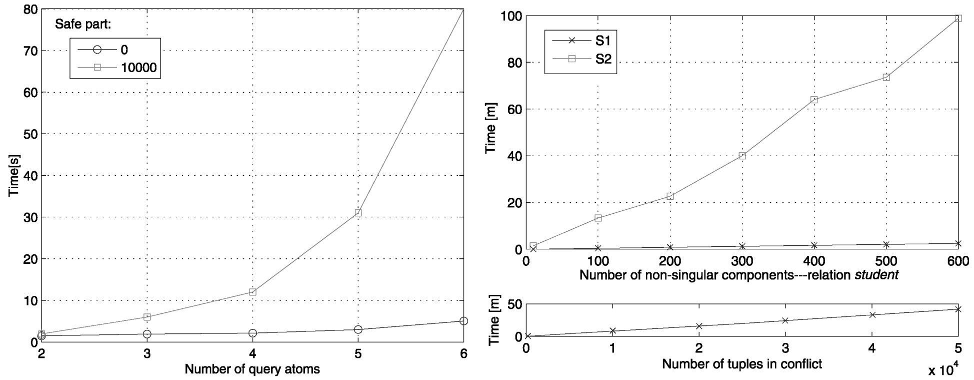
In these experiments, the safe part is fixed to 10,000 tuples (9,850 students and 150 professors), while the affected part has been generated according to two different scenarios:

(S1) The components over the relation *prof* are singular (in total we fixed 50 such components), while the components over *student* are nonsingular but decomposable (experiments have been conducted for different component sizes). Each component consists of 10 conflicting tuples. This scenario models an information system where data pertaining to professors are managed centrally, so that inconsistencies over last names may hardly emerge (last names of professors often act in practice as an identifier). However, data about students are assumed to be unreliable; they may, for instance, result from naive integration of autonomous data sources residing in different faculties or schools;

(S2) In this scenario, five nonsingular components are associated with *prof*, each one containing two tuples in conflict. Hence, this time, we assume that *prof* is also the result of some integration task (e.g., data wrapped from the Web); and, beside 50 professors in conflict over first names, five additional conflicts over last names emerge.

In all the scenarios above, repair grouping is exploited with $n = 512$.

The results for this set of experiments are reported in Figure 8(b). We note that in the lower diagram, in the case of (S1) the scaling is basically linear in the number of conflicting tuples in the relation *student* and, in fact, in the number of nonsingular components.



(a) (b)

Fig. 8. (a) Query answering over χ_f^N, $N > 2$. (b) Results for Example 5.10.

On the other hand, processing the nonsingular components associated with *prof* in (S2) causes a performance degradation depending on the number of nonsingular components of *student*. Given the bound on the number of these components, the scaling is again linear, but with faster growth rate which exponentially depends on the number of professors in conflicts over last names, as for it emerges from the upper diagram in Figure 8(b).

For a further remark on this experiment, consider again the original query Q introduced in Example 5.10, which asks for last name of professors possibly involved in family relationships with students. This query does not fall in any tractable class of queries singled out in the literature (cf. Section 7). However, according to our discussion in Section 5.2.2, the grounding technique coupled with factorization may indeed ensure tractability. Basically, for each possible last name of professor, say c, we have to compute the result of the ground query: $Q_c = \langle q, \{q(c) \leftarrow \mathit{student}(x_1, y_1, z, w_1), \mathit{prof}(c, y_2, z).\}\rangle$. Given that now we focus on precisely one component for *prof*, performances for answering Q_c are even faster than those registered for (S1). And, the global performances for evaluating Q will linearly scale in the number of distinct last names of professors in the data.

In closing this section, we would like to summarize the lesson learned from our experiments, which might give some guidelines for further investigations into consistent query answering from inconsistent database.

On the one hand, our activity has certainly provided some bad news. First, the scalability of "pure" logic-programming based approaches for consistent query answering is in many cases not suited for real-world applications. This is related to the intrinsic complexity of the problem, which is the prize to be paid for the generality of these approaches. And, second, if the data can not be factorized (e.g., by means of techniques in Section 5.2), then there is little chance to answer queries over large data sets (actually, large affected databases), even when logic-programming based approaches are used in combination with our grouping and marking techniques. On the other hand, there are two good news:

—First, in those scenarios where the size of the affected part is not very large, marking strategies can be very effective to support consistent query answering even if the data can not be factorized. Indeed, there are substantial benefits w.r.t. basic approaches where safe and affected data are not distinguished in the computation. Clearly enough, if these scenarios fall in one of the syntactically tractable classes already known in the literature, then applying the proper rewriting is definitively the best choice. However, for general classes of queries and constraints, our techniques represent a viable option, given the infeasibility of directly applying logic-programming based approaches. We point out that we experienced such kind of scenarios in applications related to data integration, and arguably they can be found in several different contexts (e.g., re-engineering of legacy systems, semantic data retrieval over the web, etc). In this regard, setting up a benchmark suite of real datasets coming from practical scenarios should be a primary goal for the community.

—Second, localization and factorization strategies may support well consistent query answering up to a large number of conflicts if the analysis of the data reveals some nice structure leading to the possibility of isolating singular and/or decomposable components. A polynomial scaling can be obtained even in some interesting settings that cannot be dealt with by approaches only focused on limiting the form of the input queries and integrity constraints (roughly, localization and factorization techniques can isolate easy instances for hard problems, on the basis of data inspection). This perspective complements previous results in the literature. In particular, if all cases in which nonsingular components are jointly decomposable, then a linear scaling can eventually be achieved for common classes of constraints and queries. However, when nondecomposable components emerge, this nice scaling tends to deteriorate (exponentially in the number of components). While this trend can not be avoided in the limit (since in the extreme case where a linear number of these components affects the data, query answering is actually co-NP-hard), the factorization strategy might be refined, in particular when a small number of nonsingular and nondecomposable components are in the data. In fact, specific elaborations in this direction constitute an interesting avenue for further research.

9. CONCLUSION

For optimizing logic-programming based query answering from inconsistent databases, we have presented a repair localization approach. In this approach, repairs are conceptually confined to a repair envelope, which intuitively comprises the part of the database affected by inconsistency, and then recombined with the unaffected (safe) part before determining the query result. We have investigated this approach in a generic framework accommodating different classes of integrity constraints (including denial constraints [Chomicki et al. 2004a]), and preference orderings for repairs from the literature (see Section 3.1). We then have discussed how this approach can be fruitfully utilized for query answering using logic programming specifications, where a logic programming engine and a DBMS are combined, such that tremendous performance gains are achieved.

While motivated by logic programming specifications, our localization results are not bound to such a setting and are, in fact, applicable to any realization of consistent query answering. Furthermore, the generic form of preferences, constraints, and repair envelopes allows us to instantiate the results to many different concrete settings in practice.

The work presented here can be extended in different directions. As for localization and query answering, our results may be extended to repair semantics based on preference orderings violating the properties in Section 3.1; for example, the one in Chomicki and Marcinkowski [2005]. Furthermore, for any negation-free query Q the intersection of the answers $Q[R]$ on all repairs is equivalent to answering it only on the repairs which are minimal under set-inclusion, that is, do not contain any other repair properly. If an ordering $\leq_D$ fails to satisfy (SIP), (DPE), and (DIS), we may characterize the repairs w.r.t. $\leq_D$ that are minimal under set inclusion as the repairs under another ordering $\leq'_D$ which satisfies these properties. An example is the ordering $R_1 \sqsubseteq_D R_2$ iff $R_1 \cap D \supseteq R_2 \cap D$ [Calì et al. 2003a, 2003b], which violates (SIP). Intuitively, in such an ordering, violations are preferably repaired by adding tuples to D, and minimally deleting tuples, in case adding tuples is not sufficient to repair inconsistency. We can use here the ordering $R_1 \sqsubseteq'_D R_2$ iff $R_1 \sqsubseteq_D R_2 \wedge (R_1 \cap D = R_2 \cap D \Rightarrow R_1 \backslash D \subseteq R_2 \backslash D)$ instead for answering Q.

Other approaches considered consistent query answering under the perspective of modifying values in the database rather than entire tuples [Wijsen 2005]. Due to the different semantics considered in these works, such repairs are not immediately captured by our framework. A study of respective extensions is left for future work.

Another extension of the results here is from a single database to a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ is the global schema, $\mathcal{S}$ is the schema of the various sources, and $\mathcal{M}$ is the mapping establishing the relationship between $\mathcal{G}$ and $\mathcal{S}$ [Lenzerini 2002]. As briefly discussed in Appendix E.1 and more in detail in Eiter et al. [2005], the results developed here can be readily adapted for a Global-As-View (GAV) setting in which $\mathcal{M}$ is given by stratified Datalog queries, and for constraints on the global schema falling in the classes considered in this paper. They can be further extended to other GAV settings, for example, as in Lembo et al. [2002]; and Calì et al. [2003b], and certain Local-As-View (LAV) settings, for example, as in Bertossi et al. [2002] and Bravo and Bertossi [2003].

In fact, most of the research reported here has been carried out within the EU project INFOMIX on advanced data integration for expressive schemas using logic programming. However, the INFOMIX system is not the implementation of all results in this paper. For more information about the project, see [Leone et al. 2005].

ACKNOWLEDGMENTS

We thank the reviewers for many helpful comments.

REFERENCES

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

Arenas, M., Bertossi, L., and Chomicki, J. 2001. Scalar aggregation in fd-inconsistent databases. In *Proceedings of the 8th International Conference on Database Theory (ICDT'01)*. Springer, 39–53.

Arenas, M., Bertossi, L., and Chomicki, J. 1999. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGACT SIGMOD Symposium on Principles of Database Systems (PODS'99)*. 68–79.

Arenas, M., Bertossi, L., and Chomicki, J. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Logic Program. 3*, 4, 393–424.

Arieli, O., Denecker, M., Nuffelen, B. V., and Bruynooghe, M. 2004. Coherent integration of databases by abductive logic programming. *J. AI. Res. 21*, 245–286.

Barceló, P. and Bertossi, L. 2003. Logic programs for querying inconsistent databases. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. 208–222.

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

Bertossi, L., Chomicki, J., Cortes, A., and Gutierrez, C. 2002. Consistent answers from integrated data sources. In *Proceedings of the 6th International Conference on Flexible Query Answering Systems (FQAS'02)*. 71–85.

Bertossi, L., Hunter, A., and Schaub, T., Eds. 2005. *Inconsistency Tolerance (result from a Dagstuhl seminar)*. Lecture Notes in Computer Science, vol. 3300, Springer.

Bouzeghoub, M. and Lenzerini, M. 2001. Introduction to the special issue on data extraction, cleaning, and reconciliation. *Inform. Syst. 26*, 8, 535–536.

Bravo, L. and Bertossi, L. 2003. Logic programming for consistently querying data integration systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*. 10–15.

Bravo, L. and Bertossi, L. 2005. Disjunctive deductive databases for computing certain and consistent answers to queries from mediated data integration systems. *J. Appl. Logic 3*, 1, 329–367.

Calì, A., Lembo, D., and Rosati, R. 2003a. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proceedings of the 22nd ACM SIGACT SIGMOD Symposium on Principles of Database Systems (PODS'03)*. 260–271.

Calì, A., Lembo, D., and Rosati, R. 2003b. Query rewriting and answering under constraints in data integration systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*. 16–21.

Celle, A. and Bertossi, L. 2000. Querying inconsistent databases: Algorithms and implementation. In *Proceedings of the International Conference on Computational Logic (CL'00)*. 942–956.

Chomicki, J. 2007. Consistent query answering: Five easy pieces. In *Proceedings of the 11th International Conference on Database Theory (ICDT'07)*, T. Schwentick and D. Suciu, Eds. Lecture Notes in Computer Science, Springer, 1–17.

Chomicki, J. and Marcinkowski, J. 2005. Minimal-change integrity maintenance using tuple deletions. *Inform. Comput. 197*, 1-2, 90–121.

Chomicki, J., Marcinkowski, J., and Staworko, S. 2004a. Computing consistent query answers using conflict hypergraphs. In *Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM'04)*. ACM Press, 417–426.

Chomicki, J., Marcinkowski, J., and Staworko, S. 2004b. Hippo: A system for computing consistent answers to a class of SQL queries. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT'04)*. Lecture Notes in Computer Science, vol. 2992, Springer, 841–844.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33*, 3, 374–425.

Eiter, T., Fink, M., Greco, G., and Lembo, D. 2003. Efficient evaluation of logic programs for querying data integration systems. In *Proceedings of the 19th International Conference on Logic Programming (ICLP'03)*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 2916, Springer, 163–177.

Eiter, T., Fink, M., Greco, G., and Lembo, D. 2005. Optimization methods for logic-based query answering from inconsistent data integration systems. Tech. rep. INFSYS RR-1843-05-05, TU Wien.

Eiter, T., Gottlob, G., and Mannila, H. 1997. Disjunctive Datalog. *ACM Trans. Data. Syst. 22*, 3, 364–418.

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

Fuxman, A., Fazli, E., and Miller, R. J. 2005. ConQuer: Efficient management of inconsistent databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 155–166.

Fuxman, A. and Miller, R. J. 2007. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci. 73*, 4, 610–635.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9*, 365–385.

Greco, G., Greco, S., and Zumpano, E. 2003. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Engin. 15*, 6, 1389–1408.

Grieco, L., Lembo, D., Ruzzi, M., and Rosati, R. 2005. Consistent query answering under key and exclusion dependencies: Algorithms and experiments. In *Proceedings of the 14th International Conference on Information and Knowledge Management (CIKM'05)*. 792–799.

Kifer, M. and Lozinskii, E. L. 1992. A logic for reasoning with inconsistency. *J. Automat. Reason. 9*, 2, 179–215.

Kowalski, R. A. and Dadri, F. 1990. Logic programming with exceptions. In *Proceedings of the 7th International Conference on Logic Programming (ICLP'90)*. 490–504.

Lembo, D., Lenzerini, M., and Rosati, R. 2002. Source inconsistency and incompleteness in data integration. In *Proceedings of the 9th International Workshop on Knowledge Representation Meets Databases (KR'02)*.

Lenzerini, M. 2002. Data integration: A theoretical perspective. In *Proceedings of the 21st ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS'02)*. 233–246.

Leone, N., Eiter, T., Faber, W., Fink, M., Gottlob, G., Greco, G., Ianni, G., Kalka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Rosati, R., Ruzzi, M., Staniszkis, W., and Terracina, G. 2005. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 915–917.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic 7*, 3, 499–562.

Lin, J. 1996. Integration of weighted knowledge bases. *AI. 83*, 2, 363–378.

Lin, J. and Mendelzon, A. O. 1998. Merging databases under constraints. *Int. J. Coop. Inform. Syst. 7*, 1, 55–76.

Nuffelen, B. V., Cortés-Calabuig, A., Denecker, M., Arieli, O., and Bruynooghe, M. 2004. Data integration using ID-logic. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04)*, A. Persson and J. Stirna, Eds. Lecture Notes in Computer Science, vol. 3084, Springer, 67–81.

Simons, P., Niemelä, I., and Soininen, T. 2002. Extending and implementing the stable model semantics. *AI. 138*, 181–234.

Staworko, S., Chomicki, J., and Marcinkowski, J. 2006. Preference-driven querying of inconsistent relational databases. In *EDBT Workshops*, T. Grust, et al., Eds. Lecture Notes in Computer Science, vol. 4254, Springer, 318–335.

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

Wijsen, J. 2005. Database repairing using updates. *ACM Trans. Datab. Syst. 30*, 3, 722–768.

Received January 2007; revised September 2007, January 2008; accepted February 2008