Computing Consistent Query Answers using Conflict Hypergraphs*

Jan Chomicki
University at Buffalo
chomicki@cse.buffalo.edu

Jerzy Marcinkowski
Wroclaw University
jma@ii.uni.wroc.pl

Slawomir Staworko
University at Buffalo
staworko@cse.buffalo.edu

Abstract

A consistent query answer in a possibly inconsistent database is an answer which is true in every (minimal) repair of the database. We present here a practical framework for computing consistent query answers for large, possibly inconsistent relational databases. We consider relational algebra queries without projection, and denial constraints. Because our framework handles union queries, we can effectively (and efficiently) extract indefinite disjunctive information from an inconsistent database. We describe a number of novel optimization techniques applicable in this context and summarize experimental results that validate our approach.

Categories: H.2.3, H.2.4, F.4.1.

General Terms: Algorithms, Languages, Theory.

Keywords: inconsistency, integrity constraints, query processing.

1. INTRODUCTION

Traditionally, the main role of integrity constraints in databases was to enforce consistency. The occurrence of integrity violations was prevented by DBMS software. However, while integrity constraints continue to express important semantic properties of data, enforcing the constraints has become problematic in current database applications. For example, in data integration systems integrity violations may be due to the presence of multiple autonomous data sources. The sources may separately satisfy the constraints, but when they are integrated the constraints may not hold. Moreover, because the sources are autonomous, the violations cannot be simply fixed by removing the data involved in the violations.

EXAMPLE 1. *Let* Student *be a relation schema with the attributes* Name *and* Address *and the key functional dependency* $Name \rightarrow Address$. *Consider the following instance of* Student*:*

Name	*Address*
Jeremy Burford	*Los Angeles*
Jeremy Burford	*New York*
Linda Kenner	*Chicago*

The first two tuples may come from different data sources, so it may be impossible or impractical to resolve the inconsistency between them. However, there is clearly a difference between the first two tuples and the third one. We don't know whether Jeremy Burford lives in Los Angeles or New York, but we do know that Linda Kenner lives in Chicago. An approach to query answering that ignores inconsistencies will be unable to make this distinction – the distinction between reliable and unreliable data. On the other hand, any approach that simply masks out inconsistent data (the first two tuples in this example) will lose indefinite information present in inconsistent databases. In this example, we know that there is a student named Jeremy Burford (existential information) and that Jeremy Burford lives in Los Angeles or New York (disjunctive information).

The above example illustrates the need to modify the standard notion of query answer in the context of inconsistent databases. We need to be able to talk about query answers that are *unaffected by integrity violations.* In [2], the notion of *consistent query answer* was proposed to achieve that objective. [2] introduced the notion of *repair*: a database that satisfies the integrity constraints and is minimally different from the original database. A *consistent answer* to a query, in this framework, is an answer present in the result of the query in *every* repair.

EXAMPLE 2. *In Example 1, there are two repairs corresponding to two different ways of restoring the consistency: either the first or the second tuple is deleted. If a query asks for all the information about students, only the tuple (Linda Kenner,Chicago) is returned as a consistent answer because it is the only tuple that is present in both repairs. On the other hand, if a query asks for the names of students living in Los Angeles or New York, then Jeremy Burford is a consistent answer.*

The framework of [2] has served as a foundation for most of the subsequent work in the area of querying inconsistent databases [3, 5, 11, 12, 13, 15, 17, 19, 23] (see [7] for a survey and an in-depth discussion). The work presented here addresses the issue of computing consistent query answers for *projection-free queries* and *denial integrity constraints.* It is shown in [13] that this task can be done in polynomial time, using the notion of *conflict hypergraph* that succinctly

*Research supported by NSF Grant IIS-0119186.

represents all the integrity violations in a given database. This line research is pursued further in the present paper.

The main contributions of this paper are as follows:

- A complete, scalable framework for computing consistent answers to projection-free relational algebra queries in the presence of denial constraints. Our approach uses a relational DBMS as a backend and scales up to large databases.
- Novel optimization techniques to eliminate redundant DBMS queries.
- Encouraging experimental results that compare our approach with an approach based on query rewriting and estimate the overhead of computing consistent query answers. No comprehensive results of this kind exist in the literature.

Because our query language includes union, our approach can extract indefinite *disjunctive* information present in an inconsistent database (see Example 1). Moreover, consistent query answers are computed in polynomial time. Other existing approaches are either unable to handle disjunction in queries [2, 12, 17] or cannot guarantee polynomial time computability of consistent query answers [3, 5, 11, 15, 19, 23]. The latter is due to the fact that those approaches rely on the computation of answers sets of logic programs with disjunction and negation – a Σ_2^p-complete problem. Only the approach of [2, 12] (which uses query rewriting) and the approach presented here scale up to large databases. Related research is further discussed in Section 6.

The plan of the paper is as follows. In Section 2, we introduce basic concepts. In Section 3, we present our approach to computing consistent answers to projection-free queries and describe its implementation in a system called Hippo. In Section 4, we describe several techniques for eliminating redundant DBMS queries, that we have implemented in Hippo. In Section 5, we discuss a number of experiments we have conducted with Hippo and query rewriting. In Section 6, we briefly discuss related work. Section 7 contains conclusions and a discussion of possible future research directions.

2. BASIC NOTIONS AND FACTS

2.1 Query languages

In this paper we work in the relational model of data. We recall that a database schema $\mathcal{S}$ is a set of relation names with attribute names and types. An instance of a database is a function that assigns a finite set of tuples to each relation name. For the purposes of this paper we consider only two fixed database domains $\mathbb{N}$ (natural numbers) and D (uninterpreted constants). We also use the natural interpretation over $\mathbb{N}$ of binary relational symbols $=, \neq, <, >$, and we assume that two constants are equal only if they have the same name. We also view I as a structure for the first-order language over the vocabulary consisting of symbols of $\mathcal{S}$, and standard built-in predicates over $\mathbb{N}$ ($=, \neq, <, >$).

In this article, we use projection-free (π-free) relational algebra expressions, defined using the following grammar:

$$E ::\equiv R \mid \sigma_X(E) \mid E \times E \mid E \cup E \mid E \setminus E.$$

$|R|$ is the arity of the relation symbol R and (unless specified otherwise) for the sake of simplicity we assume that attribute names are consecutive natural numbers. We extend this to expressions, i.e. $|E|$ is the *arity* of the expression, and $E.i$ is the reference to the i-th column resulting from the expression E (used in conditions for subexpressions). Morover, $t[i]$ is the value on the i-th position of t, $t[i, j]$ is an abbreviation for a tuple $(t[i], \ldots, t[j])$, and with $|t|$ we denote the length of the tuple t. We say that a tuple t is *compatible* with an expression E if the length of the tuple is equal to the arity of the expression, i.e. $|t| = |E|$.
For a given expression E, $\mathrm{QA}^E(I)$ is the result of evaluating E in the database instance I. In this paper we use only the set semantics of relational algebra expressions.

We also use relational calculus queries consisting of quantifier-free first-order formulas which may be open (having free variables) or ground. In fact, our approach can handle relational algebra queries that require projection, as long as they can be translated to quantifier-free relational calculus queries. That's why we can deal with the relational algebra query corresponding to the query

$$\mathit{Student}(X, '\mathrm{LosAngeles}') \vee \mathit{Student}(X, '\mathrm{NewYork}')$$

in Example 1.We also occasionally use SQL.

2.2 Repairs and consistent query answers

An integrity constraint is a consistent closed first-order formula. In this paper we consider only the class of *denial* integrity constraints of the form:

$$\forall \bar{x}_1, \ldots, \bar{x}_k . \neg \left[R_{i_1}(\bar{x}_1) \wedge \ldots \wedge R_{i_k}(\bar{x}_k) \wedge \phi(\bar{x}_1, \ldots, \bar{x}_k) \right], \tag{1}$$

where ϕ is a boolean expression consisting of atomic formulas referring to built-in predicates. The number k is called the *arity* of a constraint.
Note that, for example, functional dependencies and exclusion constraints are of the above form. Below we give another example.

EXAMPLE 3. *Consider the relation Emp with attributes Name, Salary, and Manager, with Name being the primary key. The constraint that no employee can have a salary greater that that of her manager is a denial constraint:*

$$\forall n, s, m, s', m'. \ \neg[\mathit{Emp}(n, s, m) \wedge \mathit{Emp}(m, s', m') \wedge s > s'].$$

DEFINITION 1 (CONSISTENT DATABASE). *A database instance I is* consistent *with a set of integrity constraints C if $I \models C$ (i.e., C is true in I);* inconsistent *otherwise.*

DEFINITION 2. *For a given database instance I of schema $\mathcal{S}$, its* set of facts *$\Sigma(I)$ is the set of all positive facts that hold in this database:*

$$\Sigma(I) = \{R(t) | R \in \mathcal{S} \wedge t \in I(R)\}.$$

DEFINITION 3 (DATABASE DISTANCE). *Given two instances I_1 and I_2 of the same database, the* distance *between those instances $\Delta(I_1, I_2)$ is the symmetric difference between sets of facts of those instances:*

$$\Delta(I_1, I_2) = (\Sigma(I_1) \setminus \Sigma(I_2)) \cup (\Sigma(I_2) \setminus \Sigma(I_1)).$$

DEFINITION 4 (PROXIMITY RELATION). *Given three instances I, I_1, I_2, the instance I_1 is* closer *to I than the instance I_2 if the distance between I_1 and I is contained in the distance between I_2 and I, i.e.*

$$I_1 \leq_I I_2 \iff \Delta(I, I_1) \subseteq \Delta(I, I_2).$$

Definition 5 (Database repair). *For a given instance I and set of integrity constraints C, I' is a* repair *of I w.r.t. C if I' is the closest instance to I, which is consistent with C , i.e. $I' \models C$ and I' is $\leq_I$-minimal among the instances that satisfy C.*

By $\mathrm{Rep}_C(I)$ we denote the set of all repairs of I with respect to C.

The following fact captures an important property of repairs of denial constraints: each repair is a *maximal consistent subset* of the database.

Fact 1. *If C consists only of denial constraints, then:*

$$I' \in \mathrm{Rep}_C(I) \Rightarrow \Sigma(I') \subseteq \Sigma(I).$$

Definition 6 (Core instance). *For a given instance I, its* core *w.r.t a set of integrity constraints C is an instance Core_C^I such that:*

$$\mathrm{Core}_C^I(R) = \bigcap_{I' \in \mathrm{Rep}_C(I)} I'(R).$$

For any relation R and set of integrity constraints C, if there exists a relational algebra expression Δ_C^R such that that for any instance I:

$$\mathrm{QA}^{\Delta_C^R}(I) = \mathrm{Core}_C^I(R),$$

we call Δ_C^R a core expression *of the relation R w.r.t the set of integrity constraints C.*

Fact 2. *If C is a set of denial integrity constraints, then for any $R \in \mathcal{S}$ there exists a core expression Δ_C^R of R w.r.t C.*

Example 4. *Suppose we have a table $P(A, B)$ with a functional dependency $A \rightarrow B$. The core expression for P in SQL is:*

```
SELECT * FROM P P1 WHERE NOT EXISTS (
    SELECT * FROM P P2
        WHERE P1.A = P2.A AND P1.B <> P2.B);
```

Having defined repairs, we can define consistent answers to queries. In general, the intuition is that the consistent query answer is an answer to the query in every repair. In this paper we consider consistent answers for two classes of queries.

Definition 7 (CQA for ground queries). *Given a database instance I and a set of denial integrity constraints C, we say that true (resp. false) is the consistent answer to a ground query ψ w.r.t. C in I , and we write $I \models_C \psi$, if in every repair $I' \in \mathrm{Rep}_C(I)$, $I' \models \psi$ (resp. $I' \not\models \psi$).*

Definition 8 (CQA for relational algebra). *Given a database instance I and a set of denial integrity constraints C, the set of* consistent answers *to a query E w.r.t. C in I is defined as follows:*

$$\mathrm{CQA}_C^E(I) = \bigcap_{I' \in \mathrm{Rep}_C(I)} \mathrm{QA}^E(I').$$

2.3 Conflict hypergraphs

The *conflict hypergraph* [13] constitutes a compact, space-efficient representation of all repairs of a given database instance. Note that this representation is specifically geared toward denial constraints.

Definition 9 (Conflict). *For a given integrity constraint c of form (1), a set of facts $\{R_{i_1}(t_1), \ldots, R_{i_k}(t_k)\}$, where $t_j \in I(R_{i_j})$, is a* conflict *in a database instance I if $\phi(t_1, \ldots, t_k)$. By $\mathcal{E}_{c,I}$ we denote the set of all conflicts generated by the integrity constraint c in I.*

Definition 10 (Conflict hypergraph). *For a given set of integrity constraints C and a database instance I, a* conflict hypergraph *$\mathcal{G}_{C,I}$ is a hypergraph with the set of vertices being the set of facts from the instance I, and the set of hyperedges consisting of all conflicts generated by constraints from C in I, i.e.*

$$\mathcal{G}_{C,I} = (\mathcal{V}_I, \mathcal{E}_{C,I}),\ \textit{where}\ \mathcal{V}_I = \Sigma(I),\ \textit{and}\ \mathcal{E}_{C,I} = \bigcup_{c \in C} \mathcal{E}_{c,I}.$$

Definition 11 (Maximal independent set). *For a hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the set of vertices is a* maximal independent set *if it is a maximal set that contains no hyperedge from $\mathcal{E}$.*

Fact 3. *Let I be a database instance, and C a set of denial constraints, then for any repair $I' \in \mathrm{Rep}_C(I)$, $\Sigma(I')$ is a maximal independent set M in $\mathcal{G}_{C,I}$, and vice versa.*

As shown in the following example in case of denial constraints the set of conflicts can be defined using a simple query.

Example 5. *Suppose we have a table $P(A, B)$ with a functional dependency $A \rightarrow B$. The SQL expression for selecting all conflicts from P generated by the functional constraint is:*

```
SELECT * FROM P P1, P P2
    WHERE P1.A = P2.A AND P1.B <> P2.B;
```

Definition 12 (Data complexity). *The* data complexity *of consistent answers to ground first-order queries is the complexity of determining the membership in the set $D_{C,\varphi} = \{I | I \models_C \varphi\}$, where φ is a fixed ground first-order query, and C is a fixed finite set of integrity constraints.*

We note that for a fixed set of integrity constraints, the conflict hypergraph is of polynomial size (in the number of tuples in the database instance).

3. IMPLEMENTATION

3.1 Consistent query answers

We review here the algorithm [13] for checking the consistency of ground queries in the presence of denial constraints, and then show how to use it to answer π-free queries relational algebra queries, which which correspond to open quantifier-free relational calculus queries.

We assume here that we work with a set of integrity constraints consisting only of denial constraints. The input to the algorithm consists of a ground quantifier-free formula ψ, a set of integrity constraints C, and a database instance I. We want the algorithm to answer the question whether $I \models_C \psi$.

Theorem 1. *[13] The data complexity of consistent answers to quantifier-free ground queries w.r.t a set of denial constraints is in P.*

The proof of this theorem can be found in [13] together with the corresponding algorithm that we call *HProver*. This algorithm takes the query in CNF, and a conflict hypergraph $\mathcal{G}_{C,I}$ that corresponds to the database instance I in the presence of integrity constraints C. The first step of

```
Input: $\psi = \psi_1 \wedge \ldots \wedge \psi_k$ – ground input formula in CNF,
       $\mathcal{G}_{C,I} = (\mathcal{V}_I, \mathcal{E}_{C,I})$ – conflict hypergraph of $I$ w.r.t. $C$.
1  for $i \in \{1, \ldots, k\}$ do
2      let $\neg\psi_i \equiv \neg R_{i_1}(t_1) \wedge \ldots \wedge \neg R_{i_p}(t_p) \wedge$
                  $\wedge R_{i_{p+1}}(t_{p+1}) \wedge \ldots R_{i_m}(t_m)$.
3      for $j \in \{p+1, \ldots, m\}$ do
4          if $t_j \notin I(R_{i_j})$ then
5              next $i$;
6      $B \leftarrow \{R_{i_{p+1}}(t_{p+1}), \ldots, R_{i_m}(t_m)\}$
7      for $j \in \{1, \ldots, p\}$ do
8          if $t_j \in I(R_{i_j})$ then
9              choose $e_j \in \{e \in \mathcal{E}_{C,I} | R_{i_j}(t_j) \in e\}$ nondeterm.
10             $B \leftarrow B \cup (e_j \setminus \{R_{i_j}(t_j)\})$.
11     if $B$ is independent in $\mathcal{G}_{C,I}$ then
12         return false;
13 return true;
```

Figure 1: Algorithm *HProver*

the algorithm reduces the task of determining whether *true* is the consistent answer to the query ψ to answering the same question for every conjunct ψ_i. Then each formula ψ_i is negated and the rest of the algorithm attempts to find a repair I' in which $\neg\psi_i$ is true, i.e., in which

1. $t_j \in I'(R_{i_j})$ for $(j = p+1, \ldots, m)$
2. $t_j \notin I'(R_{i_j})$ for $(j = 1, \ldots, p)$

Such a repair corresponds to a maximal independent set M in the conflict hypergraph such that:

$1'$. every of $R_{i_{p+1}}(t_{p+1}), \ldots, R_{i_m}(t_m)$ is an element of M,

$2'$. none of $R_{i_1}(t_1), \ldots, R_{i_p}(t_p)$ is an element of M.

If the algorithm succeeds in building an independent set satisfying the properties $1'$ and $2'$, such a set can be extended to a maximal one which also satisfies those properties. That means that there is a repair in which $\neg\psi_i$, and thus also $\neg\phi$, is true. If the algorithm does not succeed for any i, $i = 1, \ldots, k$, then *true* is the consistent answer to ϕ.

The condition $1'$ is satisfied by simply *including* the appropriate facts in M. The condition $2'$ is satisfied by *excluding* the appropriate facts from M. A fact can be excluded if it is not in $\Sigma(I)$ or if it belongs to a hyperedge whose remaining elements are already in M.

3.2 Finding an envelope

Any relational algebra expression E can be translated to a corresponding first-order formula $\psi_E(\bar{x})$ in a standard way. Since we consider only π-free algebra expressions, the formula $\psi_E(\bar{x})$ is quantifier-free. To be able to use *HProver*, we have to *ground* this formula, i.e., find an appropriate set of bindings for the variables in the formula. This will be done by evaluating an *envelope* query over the database. An envelope query should satisfy two properties: (1) it should return a *superset* of the set of consistent query answers for every database instance, and (2) it should be easily constructible from the original query. The result of evaluating an envelope query over a given database will be called an *envelope*.

Suppose K_E is an envelope query for a query E. We have that

$$\mathrm{CQA}_C^E(I) = \{\bar{t} \in \mathrm{QA}^{K_E}(I) \mid I \models_C \psi_E(\bar{t})\}.$$

If an expression E does not use the difference operator (and thus is a monotonic expression), E itself is an envelope query, as stated by the following lemma:

Lemma 1. *For any monotonic relational expression E, the following holds:*

$$\mathrm{CQA}_C^E(I) \subseteq \mathrm{QA}^E(I).$$

However when E is not monotonic, then the set of consistent query answers may contain tuples not contained in $\mathrm{QA}^E(I)$. That kind of a situation is shown in the example below.

Example 6. *Suppose we have two relations $R(A,B)$ and $S(A,B,C,D)$, and we have functional dependency over $R: A \rightarrow B$. In case when $I(R) = \{(1,2),(1,3)\}$, and $I(S) = \{(1,2,1,3)\}$, the set of answers to the query*

$$E = S \setminus (R(A_1,B_1) \bowtie_{B_1 \neq B_2} R(A_2,B_2))$$

is $\emptyset$, while the set of consistent query answers is $\{(1,2,1,3)\}$.

To obtain the expression for an envelope, we define two operators F and G by mutual recursion. The operator F defines the envelope by overestimating the set of consistent answers. The auxiliary operator G underestimates the set of consistent answers.

Definition 13. *We define the operators F and G recursively:*

$$\begin{aligned}
F(R) &= R,\\
F(E_1 \cup E_2) &= F(E_1) \cup F(E_2),\\
F(E_1 \setminus E_2) &= F(E_1) \setminus G(E_2),\\
F(E_1 \times E_2) &= F(E_1) \times F(E_2),\\
F(\sigma_\chi(E)) &= \sigma_\chi(F(E)),\\
G(R) &= \Delta_C^R,\\
G(E_1 \cup E_2) &= G(E_1) \cup G(E_2),\\
G(E_1 \setminus E_2) &= G(E_1) \setminus F(E_2),\\
G(E_1 \times E_2) &= G(E_1) \times G(E_2),\\
G(\sigma_\chi(E)) &= \sigma_\chi(G(E)).
\end{aligned}$$

Because C consist only of denial constraints, Fact 2 guarantees that the expression Δ_C^R exists, and therefore the operators are well defined. The pair of operators (F,G) has the following properties:

Lemma 2. *For any π-free relational algebra expression E:*

$$\mathrm{QA}^{G(E)}(I) \subseteq \mathrm{QA}^E(I) \subseteq \mathrm{QA}^{F(E)}(I), \textit{ and}$$
$$\mathrm{CQA}_C^{G(E)}(I) \subseteq \mathrm{CQA}_C^E(I) \subseteq \mathrm{CQA}_C^{F(E)}(I).$$

Lemma 3. *For any π-free relational algebra expression E:*

$$\forall I' \in \mathrm{Rep}_C(I).\, \mathrm{QA}^{G(E)}(I) \subseteq \mathrm{QA}^E(I') \subseteq \mathrm{QA}^{F(E)}(I)$$

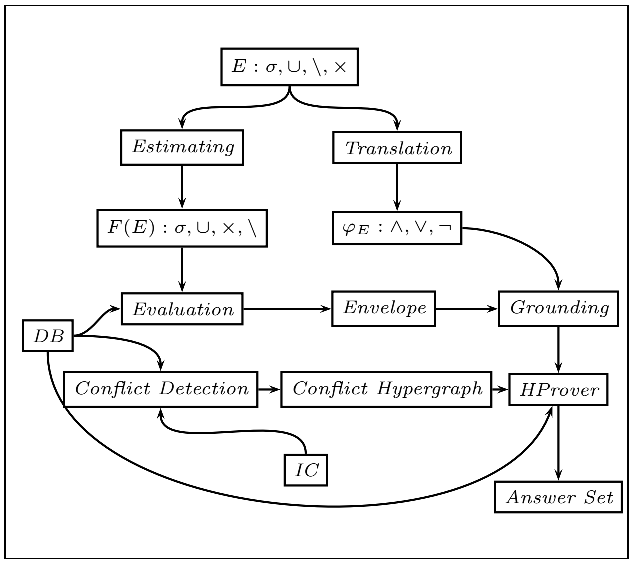
With those two lemmas we can prove the following theorem.

Theorem 2. *If C contains only denial constraints, then for any π-free relational algebra expression E the following holds for every database instance I:*

$$\mathrm{QA}^{G(E)}(I) \subseteq \mathrm{CQA}_C^E(I) \subseteq \mathrm{QA}^{F(E)}(I).$$

3.3 The system Hippo

We have implemented a system called Hippo for finding consistent answers to π-free relational algebra queries. The data is stored in an RDBMS (in our case, PostgreSQL). The flow of data in Hippo is shown in Figure 2. The only



$E : \sigma, \cup, \setminus, \times$
Estimating
Translation
$F(E) : \sigma, \cup, \times, \setminus$
$\varphi_E : \wedge, \vee, \neg$
Evaluation
Envelope
Grounding
DB
Conflict Detection
Conflict Hypergraph
HProver
IC
Answer Set

Figure 2: Data flow in Hippo

output of this system is the *Answer Set* consisting of the consistent answers to the input query E with respect to a set of integrity constraints *IC* in the database instance *DB*. Before processing any input query, the system performs *Conflict Detection*, and creates the *Conflict Hypergraph*. We assume that the number of conflicts is small enough to allow us to store the hypergraph in main memory. We keep in main memory only the set of hyperedges corresponding to conflicts in database. The set of all the vertices represents the entire contents of the database and thus may be too big to fit in main memory. In this way, we guarantee that our approach is *scalable*.

The processing of a query E consists of *Estimating* it to an envelope query $F(E)$ that after *Evaluation* by an RDBMS gives us the *Envelope*. Also, the system performs *Translation* of the input query E to a corresponding first-order logic formula φ_E. Now, for every tuple from the *Envelope* we perform *Grounding* of φ_E. Having now a first-order ground query we can check if *true* is the consistent answer to this query using *HProver*. Depending on the result of this check we return the tuple or not. It's important to notice here that because the hypergraph is stored in main memory, *HProver* doesn't need any immediate knowledge of the integrity constraints (no arrow from *IC* to *HProver*). This is because in *HProver* the independence of constructed sets B is being checked only for sets of vertices that are contained in the database, and if such vertices are in any conflict, it is registered in the hypergraph. *HProver* makes, however, database accesses to check tuple membership in database relations.

4. OPTIMIZATIONS

The previous section showed how to build a system for computing consistent query answers. But even though we have decided to store the conflict hypergraph in main memory, we still have to perform tuple membership checks (steps 4 and 8 in the *HProver* algorithm). To check if a tuple is present in a given table, we execute a simple membership query. For every tuple from the envelope we have to perform several tuple checks (depending on the complexity of the query). Executing any query is usually a costly operation in the database context. Therefore tuple membership checks are a significant factor in the algorithm execution time.

In this section we address the problem of eliminating tuple membership checks. We propose two improvements:

1. The first infers information about the tuples present in the database from the current envelope tuple. That makes it possible to answer some tuple checks without interrogating the database.
2. The second supplements the first by extending the envelope expression so that we can find the results of all relevant tuple checks without executing any membership query.

4.1 Knowledge gathering

In this section we address the problem of answering tuple checks.

Definition 14 (Relevant facts). *For a given π-free expression E and a tuple t compatible with E, the set* $\mathrm{TC}(E,t)$ *of* relevant facts *is defined recursively:*

$$\begin{aligned}\mathrm{TC}(R,t) &= \{R(t)\},\\ \mathrm{TC}(E_1 \cup E_2, t) &= \mathrm{TC}(E_1,t) \cup \mathrm{TC}(E_2,t),\\ \mathrm{TC}(E_1 \setminus E_2, t) &= \mathrm{TC}(E_1,t) \cup \mathrm{TC}(E_2,t),\\ \mathrm{TC}(E_1 \times E_2, (t_1,t_2)) &= \mathrm{TC}(E_1,t_1) \cup \mathrm{TC}(E_2,t_2),\\ \mathrm{TC}(\sigma_\chi(E),t) &= \mathrm{TC}(E,t).\end{aligned}$$

The set of facts $\mathrm{TC}(E,t)$ consists of all facts that *HProver* may need when working with the query $\phi_E(t)$ (we conjecture that the same set of facts will be needed by any practical checker of consistent query answers for quantifier-free queries). In the following example we show that the tuple t itself may carry information that can be used to derive some relevant facts.

Example 7. *Recall that relation attributes are named by natural numbers. Assume that we have two tables $R(1,2)$, $P(1,2)$ and a query $E = F(E) = \sigma_{1=a}(R \times (R \cup P))$. Suppose that a tuple $t = (a,b,c,d)$ is the only result of the evaluation of $F(E)$ in a database instance I. The set of relevant facts is $\mathrm{TC}(E,t) = \{R(a,b), R(c,d), P(c,d)\}$. A natural consequence of the semantics of relational algebra expressions is that $t \in \mathrm{QA}^{\sigma_{1=a}(R\times(R\cup P))}(I)$ implies $(a,b) \in I(R)$. We can use this information to avoid performing some membership queries. At the same time the tuple t itself doesn't carry enough information to decide whether (c,d) belongs to either $I(R)$, $I(P)$, or both of them.*

We call the process of inferring the information from result of the evaluation of a query *knowledge gathering*. Formally, we define the set of derived facts in the following way:

Definition 15 (Knowledge gathering). *For a given π-free expression E and a tuple t compatible with E*

we define the set KG *recursively:*

$$\begin{aligned}
\mathrm{KG}(R,t) &= \{R(t)\},\\
\mathrm{KG}(E_1 \cup E_2, t) &= \mathrm{KG}(E_1,t) \cap \mathrm{KG}(E_2,t),\\
\mathrm{KG}(E_1 \setminus E_2, t) &= \mathrm{KG}(E_1,t),\\
\mathrm{KG}(\sigma_\chi(E),t) &= \mathrm{KG}(E,t),\\
\mathrm{KG}(E_1 \times E_2,(t_1,t_2)) &= \mathrm{KG}(E_1,t_1) \cup \mathrm{KG}(E_2,t_2).
\end{aligned}$$

We note here that the cardinality of the set of facts inferred with KG is linear in the size of the query and doesn't depend on the value of the tuple t. Now we state the main property of KG.

THEOREM 3 (SOUNDNESS OF KG). *Given a database instance I and a π-free expression E*

$$\forall t \in \mathrm{QA}^{F(E)}(I).\forall R(t') \in \mathrm{TC}(E,t).\quad R(t') \in \mathrm{KG}(E,t) \Rightarrow I \models R(t').$$

Knowledge gathering is also complete in the case of $\{\sigma, \times\}$-expressions, i.e. it derives all relevant facts that hold in the database I.

THEOREM 4 (COMPLETENESS OF KG FOR $\{\sigma,\times\}$). *Given a database I and any $\{\sigma,\times\}$-query E.*

$$\forall t \in \mathrm{QA}^{F(E)}(I).\forall R(t') \in \mathrm{TC}(E,t).\quad I \models R(t') \Rightarrow R(t') \in \mathrm{KG}(E,t).$$

4.2 Extended knowledge gathering

In general, when the expression translates to a disjunctive query we need to extend the query so that the resulting tuple carries some additional information allowing us to derive all relevant facts. The extended approach described in detail below is illustrated first by the following example.

EXAMPLE 8. *For the previously considered expression $E = \sigma_{1=a}(R \times (R \cup P))$ the extended approach constructs the expression $\sigma_{1=a}(R \times (R \cup P)) \overset{3,4}{\longleftarrow} R \overset{3,4}{\longleftarrow} P$, where $\leftarrow$ is the left outer join operator*[1]*. Suppose now, $I(R) = \{(a,b),(e,f)\}$ and $I(P) = \{(c,d),(e,f)\}$. Then the evaluation of the extended envelope expression yields the following:*

$\sigma_{1=a}(R \times (R \cup P)) \overset{3,4}{\longleftarrow} R \overset{3,4}{\longleftarrow} P$							
a	b	a	b	a	b	$\perp$	$\perp$
a	b	c	d	$\perp$	$\perp$	c	d
a	b	e	f	e	f	e	f

Now, consider the tuple $(a,b,c,d,\perp,\perp,c,d)$. We can decompose it into two parts (a,b,c,d) and $(\perp,\perp,c,d)$. The first part is simply the tuple from the envelope $F(E)$, and it can be used to infer the fact $R(a,b)$. The second part allows us to make two other important inferences. Namely, $(c,d) \notin I(R)$ and $(c,d) \in I(P)$.

Our goal is to minimally extend the expression so that we can derive all relevant facts. In order to find what information is not guaranteed to be gathered from evaluation of the envelope expression, we generalize the definitions of KG and TC to *non-ground* tuples consisting of distinct variables.

[1]For clarity we simplify the notion of the outer join condition. When writing $S \overset{3,4}{\longleftarrow} T$ we mean $S \xleftarrow{S.3=T.1 \wedge S.4=T.2} T$, and we assume the left join operator is left associative

DEFINITION 16 (COMPLEMENTARY SET). *For a given π-free expression E, the* complementary set $\Gamma(E)$ *is defined as follows:*

$$\Gamma(E) = \mathrm{TC}(E,\bar{x}) \setminus \mathrm{KG}(E,\bar{x}),$$

where $\bar{x} = (x_1,\ldots,x_{|E|})$.

EXAMPLE 9. *Taking again under consideration the expression $E = \sigma_{1=a}(R \times (R \cup P))$ and $\bar{x} = (x_1,\ldots,x_4)$ we have:*

$$\begin{aligned}
\mathrm{TC}(E,\bar{x}) &= \{R(x_1,x_2), P(x_3,x_4), R(x_3,x_4)\},\\
\mathrm{KG}(E,\bar{x}) &= \{R(x_1,x_2)\}.
\end{aligned}$$

$R(x_1,x_2) \in \mathrm{TC}(E)$ means that for any tuple (t_1,t_2,t_3,t_4) from the evaluation of the envelope expression for E, HProver *may perform the tuple check $R(t_1,t_2)$. We have also $R(x_1,x_2) \in \mathrm{KG}(E)$ and therefore we are able to answer this check using knowledge gathering. On the other hand $R(x_3,x_4) \in \mathrm{TC}(E)$ means that for* HProver *may perform a tuple check $R(t_3,t_4)$. Since we don't have that $R(x_3,x_4) \in \mathrm{KG}(E)$ we cannot guarantee that we can answer tuple checks $R(t_3,t_4)$ without executing a membership query on the database, even though we are able to answer tuple checks $R(t_1,t_2)$. The complementary set for the discussed expression is:*

$$\Gamma(E) = \{R(x_3,x_4), P(x_3,x_4)\}.$$

Analogous examples can be used to show that the simple knowledge gathering is not sufficient to avoid membership checks when processing expressions with the difference operator. Next, we extend the envelope expression so that it evaluation provides us with all information sufficient to answer the tuple checks.

DEFINITION 17 (EXTENDED ENVELOPE EXPRESSION). *For a given π-free expression E the extended envelope expression is defined as follows:*

$$H(E) = F(E) \underset{R(x_i,\ldots,x_{i+|R|-1}) \in \Gamma(E)}{\xleftarrow{\bigwedge_{j=1}^{|R|} E.(i+j-1)=R.j}} R.$$

The notation means that we have as many outer joins as there are elements in $\Gamma(E)$. They can appear in any order. We also define the following auxiliary expression:

$$S(E) = E \times \bigtimes_{R(x_i,\ldots,x_{i+|R|-1}) \in \Gamma(E)} R.$$

For both $H(E)$ and $S(E)$ the elements of $\Gamma(E)$ need to be considered in the same order.

Using outer joins results in a natural one-to-one correspondence between the tuples from the evaluation of the extended envelope expression and the tuples from the original envelope.

FACT 4. *For a given database instance I and π-free expression E, the map $t \mapsto t[1,|E|]$ is a one-to-one map of $\mathrm{QA}^{H(E)}(I)$ onto $\mathrm{QA}^{F(E)}(I)$.*

Extending knowledge gathering to null tuples $\mathrm{KG}(R,(\perp,\ldots,\perp)) = \emptyset$ allows us to state that using the extended envelope expression we can determine correctly all relevant facts without querying the database.

Theorem 5 (Soundness, completeness of ext. KG). *For any database instance I and a π-free expression E the following holds:*

$$\forall t \in \mathrm{QA}^{H(E)}(I).\forall R(t') \in \mathrm{TC}(E, t[1, |E|]).$$
$$R(t') \in \mathrm{KG}(S(E), t) \iff I \models R(t').$$

We note that in the case of $\{\sigma, \times\}$-expressions this approach doesn't unnecessarily extend the expression.

4.3 Other possibilities of optimizations

4.3.1 Negative knowledge gathering

Knowledge gathering KG (as defined in Section 4.1) is complete only for queries that translate to a conjunction of positive literals. However, it is possible to come up with a construction that will be complete for queries that translate to a conjunction of positive as well as negative literals. The following example presents this idea.

Example 10. *Suppose we have tables $R(1, 2)$ and $P(1, 2)$ and a set of constraints C. For the query $E = R\backslash P$, we have $F(E) = R \setminus \Delta_C^P$. Take any tuple $t \in \mathrm{QA}^{F(E)}(I)$ for some instance I. We can easily conclude that $t \in I(R)$. Also, we can say that $t \notin \mathrm{QA}^{\Delta_C^P}(I)$. Having this and hypergraph $\mathcal{G}_{C,I} = (\mathcal{V}_I, \mathcal{E}_{C,I})$ we can easily find if $t \in I(P)$. Namely, if there exists an edge $e \in \mathcal{E}_{C,I}$ that $P(t) \in e$, then $t \in I(P)$. And if the vertex $P(t)$ is not involved in any conflict in $\mathcal{E}$ then $t \notin I(P)$.*

Reasoning of that sort cannot be applied to a query $E = R \times R \setminus P \times P$. Given a tuple $t = (t_1, t_2)$ from the envelope we know that $t_1, t_2 \in R$, but the fact $(t_1, t_2) \notin \mathrm{QA}^{\Delta_C^P \times \Delta_C^P}(I)$ doesn't imply that $t_1 \notin \mathrm{QA}^{\Delta_C^P}(I)$ or $t_2 \notin \mathrm{QA}^{\Delta_C^P}(I)$. And therefore we are not able to find if $t_1 \notin I(P)$ or $t_2 \notin I(P)$.

This mechanism hasn't been included in the tested implementation yet. Implementing only positive knowledge gathering allows us to better observe the benefits of extending the envelope expression.

We notice here that the query rewriting approach to computing consistent query answers described in [2] works also only for queries that are conjunctions of literals. However, as shown below, our approach leads to faster computation of consistent answers than query rewriting.

4.3.2 Intersection

Another possible venue of optimization comes from directly implementing derived operators of relational algebra. For example, for intersection the appropriate extensions of the operators F and G are very simple:

$$F(E_1 \cap E_2) = F(E_1) \cap F(E_2),\ G(E_1 \cap E_2) = G(E_1) \cap G(E_2).$$

Now $R \cap P$ is equivalent to $R \setminus (R \setminus P)$ but $F(R \cap P) = R \cap P$ is not equivalent to $F(R \setminus (R \setminus P)) = R \setminus (\Delta_C^R \setminus P)$. Thus the envelope constructed by the operator F becomes sensitive to the way the original query is formulated.

5. EXPERIMENTAL RESULTS

5.1 The setting for the experiments

Among available methods for computing consistent query answers, only the query rewriting technique [2] seems to be feasible for large databases. This is why in this work we compare the following engines:

SQL An engine that executes the given query on the underlying RDBMS, and returns the query result. This method doesn't return consistent query answers, but provides a baseline to observe the overhead of computing consistent query answers using the proposed methods.

QR Using the SQL engine, we execute the rewritten query constructed as decribed in [2]. More details on this approach can be found in Section 6.

KG This method constructs the basic envelope expression and uses knowledge gathering, as described in Section 4.1.

ExtKG This engine constructs the extended envelope expression (Section 4.2) and uses extended knowledge gathering.

5.1.1 Generating test data

Every test was performed with the database containing two tables P and Q, both having three attributes X, Y, Z. For the constraints, we took a functional dependency $X \rightarrow Z$ in each table. The test databases had the following parameters:

- n : the number of base tuples in each table,
- m : the number of additional conflicting tuples,

and had both tables constructed in the following way:

1. Insert n different base tuples with X and Z being equal and taking subsequent values $0, \ldots, n-1$, and Y being randomly drawn from the set $\{0, 1\}$.
2. Insert m different conflicting tuples with X taking subsequent values $\{0, \lceil n/m \rceil, \lceil 2 * n/m \rceil, \ldots, \lceil (m-1) * n/m \rceil\}$, $Z = X + 1$, and Y being randomly drawn from the set $\{0, 1\}$.

In addition, we define auxiliary tables (P_{core} and Q_{core}) containing only non-conflicting tuples from the base tables (resp. P and Q). Those table were used as materialized views of the core expressions (Δ_C^P and Δ_C^Q).

Example 11. *We show how a table P with $n = 4$ and $m = 2$ can be generated:*

1. *First we insert the base tuples* $(0, 1, 0), (1, 0, 1), (2, 0, 2), (3, 1, 3)$ *into P*
2. *Then we insert the following conflicting tuples* $(0, 1, 1), (2, 0, 3)$ *into P*
3. *P_{core} will hold the following tuples* $(1, 0, 1), (3, 1, 3)$.

In every table constructed in such a way the number of tuples is $n + m$, and the number of conflicts is m.

5.1.2 The environment

The implementation is done in Java2, using PostgreSQL (version 7.3.3) as the relational backend. All test have been performed on a PC with a 1.4GHz AMD Athlon processor under SuSE Linux 8.2 (kernel ver. 2.4.20) using Sun JVM 1.4.1.

5.2 Test results

Testing a query with a given engine consisted of computing the consistent[2] answers to the query and then iterating over the results. Iteration over the result is necessary, as the subsequent elements of the consistent query answer set are computed by Hippo in a lazy manner (this allows us to process results bigger than the available main memory). Every test has been repeated three times and the median taken. Finally, we note that the cost of computing the conflict hypergraph, which is incurred only once per session, is ignored while estimating the time of the query evaluation. We take a closer look at the time required for hypergraph construction in Section 5.2.3.

5.2.1 Simple queries

We first compared performance of different engines on simple queries: join, union, and difference. Because we performed the tests for large databases, we added a range selection to the given query to obtain small query results, factoring out the time necessary to write the outputs. As parameters in the experiments, we considered the database size, the conflict percentage, and the estimated result size.

Figure 3 shows the execution time for join as a function of the size of the database. In the case of $\{\sigma, \times\}$-expressions (thus also joins), the execution times of **KG**, **ExtKG** and **SQL** are essentially identical. Since no membership queries have to be performed, it means that for simple queries the work done by *HProver* for all tuples is practically negligible.

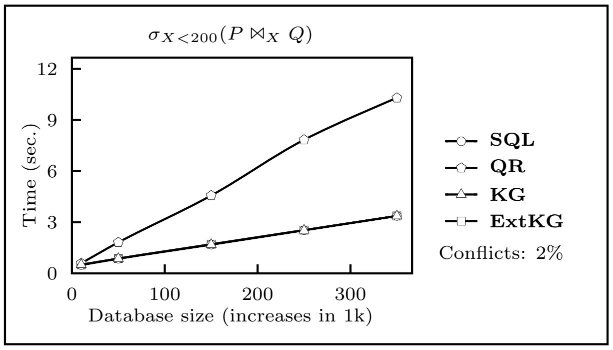


Figure 3: Execution time for join.

Figure 4 contains the results for union. It shows that basic knowledge gathering **KG** is not sufficient to efficiently handle union. The cost of performing membership queries for all tuples is very large. Note that query rewriting is not applicable to union queries.

Figure 5 contains the results for set difference (the execution time for **KG** was relatively much larger than values of other solutions and in order to increase readability it has not been included on this figure). Here the execution time is a function of the percentage of conflicts. We note that **ExtKG** performs as well as **QR** and both are approximately twice slower than **SQL**.

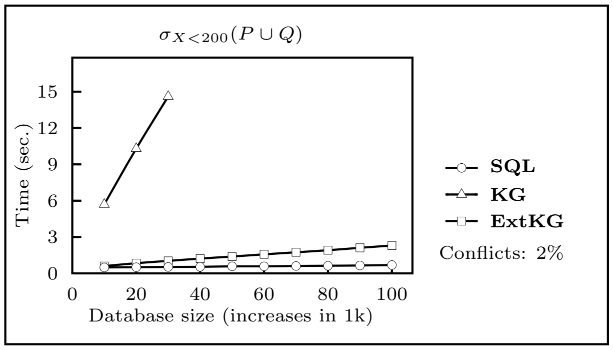


Figure 4: Execution time for union.

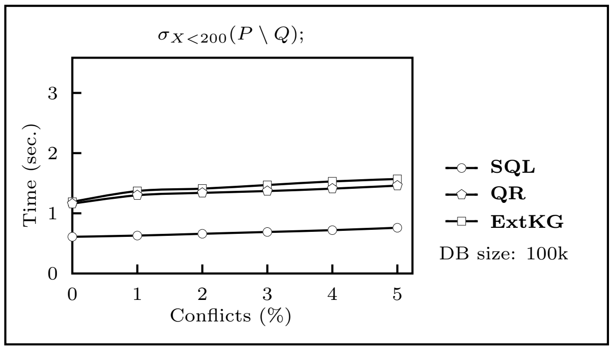


Figure 5: Execution time for difference.

5.2.2 Complex queries

In order to estimate the cost of extending the envelope we considered a complex union query

$$\sigma_{X<d}(P \bowtie_X Q \bowtie_X P \bowtie_X Q \cup Q \bowtie_X P \bowtie_X Q \bowtie_X P),$$

with d being a parameter that will allow us to control the number of tuples processed by each engine. To assure no membership queries will be performed, we have to add 8 outer joins. The main goal was to compare two versions of knowledge gathering: **KG** and **ExtKG**. We have also included the results for **SQL**. (It should be noted here that this query has common subexpressions and RDBMS might use this to optimize the query evaluation plan. PostgreSQL, however, does not perform this optimization.)

As we can see in Figure 6, **KG** outperforms **ExtKG** only in the case when the number of processed tuples is small. As the result size increases, the execution time of **ExtKG** grows significantly slower than that of **KG**. We notice also that **ExtKG** needs 2–3 times more time than **SQL** but the execution times of both grow in a similar fashion.

5.2.3 Hypergraph computation

The time of constructing the hypergraph is presented on Figure 7). It depends on the total number of conflicts and the size of the database.

It should be noticed here that the time of hypergraph construction consists mainly of the execution time of conflict detection queries. Therefore the time of hypergraph computation depends also on the number of integrity constraints and their arity.

[2]Except when using **SQL** engine.

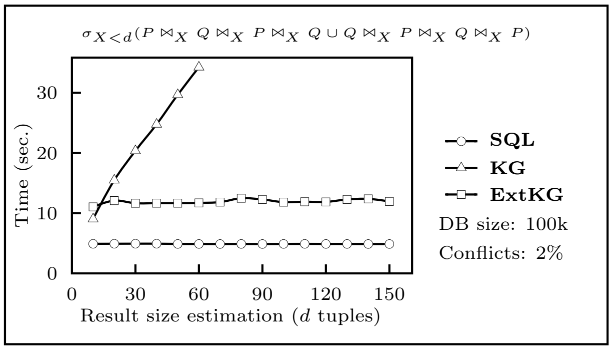


Figure 6: Impact of the result size.

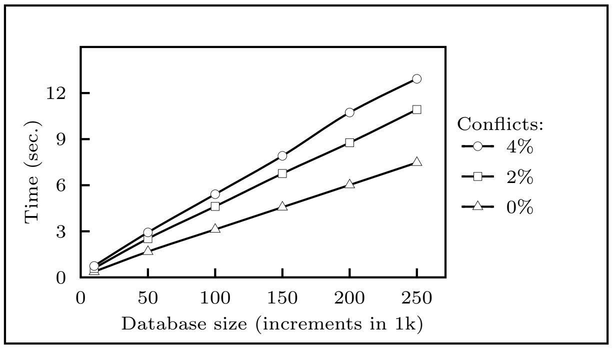


Figure 7: Hypergraph computation time

6. RELATED WORK

The discussion of related work here is very brief and focuses mainly on the most recent research. For a comprehensive discussion, please see [7].

Bry [10] was the first to note that the standard notion of query answer needs to be modified in the context of inconsistent databases and to propose the notion of a *consistent query answer*. Bry's definition of consistent query answer is based on provability in minimal logic and expresses the intuition that the part of the database instance involved in an integrity violation should not be involved in the derivation of consistent query answers. This is not quite satisfactory, as one would like to have a semantic, model-theoretic notion of consistent query answer that parallels that of the standard notion of query answer in relational databases. Moreover, the data involved in an integrity violation is not entirely useless and reliable indefinite information can often be extracted from it, as seen in Example 1.

Query rewriting [2, 12] rewrites the original query Q to another query Q' with the property that the set of all the answers to Q' in the original database is equal to the set of consistent answers to Q in that database. When applicable, this approach provides an easy way to compute consistent query answers, as the rewritten query Q' can typically be evaluated using the same query engine as the query Q. Because the query Q is rewritten independently of the database, the existence of a rewriting shows that requesting consistent query answers instead of the regular ones does not increase data complexity. However, query rewriting has been found to apply only to restricted classes of queries: the $\{\sigma, \times, \setminus\}$-subset [2] or the $\{\sigma, \pi\}$-subset [13] of the relational algebra. No method is presently known to rewrite queries with projection considered together with the binary operators, or union. Also, the class of constraints is limited to binary universal constraints [2] or single functional dependencies [13]. The line of research from [2] is continued in [17] where a class of tractable conjunctive queries, based on generalized perfect matching, is identified. It is proved that the consistent answers to queries in this class cannot be obtained by query rewriting. We note here that the nonexistence of query rewriting for conjunctive queries follows also from the fact that computing consistent query answers for such queries is a co-NP-complete problem [4, 13]. This is because the rewritten query is first-order and thus can be evaluated in AC^0, while known NP-complete problems like SAT are not in AC^0.

Several different approaches have been developed to specify all repairs of a database as a logic program with disjunction and classical negation [3, 6, 15, 18, 19, 22]. Such a program can then be evaluated using an existing system like `dlv` [14]. These approaches have the advantage of generality, as typically arbitrary first-order queries and universal constraints (or even some referential integrity constraints [3]) can be handled. However, the generality comes at a price: The classes of logic programs used are Π_2^p-complete. Therefore, the approaches based on logic programming are unlikely to work for large databases. The paper [15] proposes several optimizations that are applicable to logic programming approaches. One is localization of conflict resolution, another - encoding tuple membership in individual repairs using bit-vectors, which makes possible efficient computation of consistent query answers using bitwise operators. However, it is known that even in the presence of one functional dependency there may be *exponentially* many repairs [4]. With only 80 tuples involved in conflicts, the number of repairs may exceed 10^{12}! It is clearly impractical to efficiently manipulate bit-vectors of that size.

[11] describes several possible definitions of repair, including Definition 5, and analyzes the complexity of computing consistent query answers under those definitions. Key and inclusion dependencies are considered. The computational approaches proposed are based on combinations of repair enumeration and chase computation [1]. New tractability results are obtained for classes of databases that satisfy key constraints but may violate inclusion dependencies.

Presently, our approach requires that the integrated database be materialized at a single site. It remains to be seen if it can be generalized to a scenario where data is pulled from different sites during the evaluation of queries rewritten using, for example, the LAV approach [20]. This problem has been considered in the context of a logic-program-based approach to the computation of consistent query answers [8, 9] but, as explained earlier, such an approach does not scale up to large databases.

A new scenario for data integration, *data exchange*, has been recently proposed [16]. In this scenario, a target database is materialized on the basis of a source database using source-to-target dependencies. In the presence of target integrity constraints, a suitable consistent target database may not exist. This is a natural context for the application of the concepts of repair and consistent query answer. However, [16] does not consider the issue of the inconsistency of target databases. [11] addresses the problem of consistent query answering in a restricted data exchange setting.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a practical, scalable framework for computing consistent query answers for large databases. We have also described a number of novel optimization techniques applicable in this context and summarized experimental results that validate our approach.

The approach, however, has a number of limitations. Only projection-free relational algebra queries and denial integrity constraints are currently supported. Adding projection to the query language is a difficult issue because the complexity of computing consistent query answers becomes in that case co-NP-complete [4, 13]. So, unless P=NP, we cannot hope for computing consistent query answers efficiently for arbitrary conjunctive queries and arbitrary database instances. However, the evaluation of queries with projection can make use of the conflict hypergraph representation of all repairs, and of the operators F and G introduced in Section 3. Moreover, we expect to be able to compute consistent answers to queries with projection in polynomial time if conflict hypergraphs are suitably restricted. We hope that such restrictions can be translated into corresponding restrictions on database instances and integrity constraints.

In [4], we have studied scalar aggregation queries in the presence of functional dependencies, also making use of conflict graphs. It remains to be seen whether the techniques developed in [4] can be combined with those of the present paper.

Going beyond denial constraints appears challenging, too. Essentially, integrity violations of denial constraints are due to the *presence* of some facts in the database, and thus can be compactly represented using the conflict hypergraph. If arbitrary universal constraints, for example tuple-generating dependencies [1, 21], are allowed, constraint violations may be due to the simultaneous presence and *absence* of certain tuples in the database. It is not clear how to construct in this case a compact representation of all repairs that can be used for the computation of consistent query answers. Also, repairs are no longer guaranteed to be subsets of the original database but can contain additional tuples. If referential integrity is to be captured, constraints have to contain existentially quantified variables, which leads to the undecidability of consistent query answers [11]. Only in very restricted cases this problem has been shown to be tractable [11, 13].

Another avenue of further research involves using preferences to reduce the number of repairs and consequently make the computation of consistent query answers more efficient. For example, in data integration, we may have a preference for certain sources or for more recent information.

The issue of benchmarking systems that compute consistent query answers requires more work. It would be desirable to design mechanisms that generate inconsistent databases in a systematic way and to perform more extensive experimental comparisons between implemented systems.

8. REFERENCES

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

[2] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.

[3] M. Arenas, L. Bertossi, and J. Chomicki. Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, 3(4–5):393–424, 2003.

[4] M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar Aggregation in Inconsistent Databases. *Theoretical Computer Science*, 296(3):405–434, 2003.

[5] M. Arenas, L. Bertossi, and M. Kifer. Applications of Annotated Predicate Calculus to Querying Inconsistent Databases. In *International Conference on Computational Logic*, pages 926–941. Springer-Verlag, LNCS 1861, 2000.

[6] P. Barcelo and L. Bertossi. Logic Programs for Querying Inconsistent Databases. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 208–222. Springer-Verlag, LNCS 2562, 2003.

[7] L. Bertossi and J. Chomicki. Query Answering in Inconsistent Databases. In J. Chomicki, R. van der Meyden, and G. Snake, editors, *Logics for Emerging Applications of Databases*, pages 43–83. Springer-Verlag, 2003.

[8] L. Bertossi, J. Chomicki, A. Cortes, and C. Gutierrez. Consistent Answers from Integrated Data Sources. In *International Conference on Flexible Query Answering Systems (FQAS)*, pages 71–85, Copenhagen, Denmark, October 2002. Springer-Verlag.

[9] L. Bravo and L. Bertossi. Logic Programs for Consistently Querying Data Integration Systems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 10–15, 2003.

[10] F. Bry. Query Answering in Information Systems with Integrity Constraints. In *IFIP WG 11.5 Working Conference on Integrity and Control in Information Systems*, pages 113–130. Chapman &Hall, 1997.

[11] A. Cali, D. Lembo, and R. Rosati. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 260–271, 2003.

[12] A. Celle and L. Bertossi. Querying Inconsistent Databases: Algorithms and Implementation. In *International Conference on Computational Logic*, pages 942–956. Springer-Verlag, LNCS 1861, 2000.

[13] J. Chomicki and J. Marcinkowski. Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, 2004. To appear. Earlier version: Technical Report cs.DB/0212004, arXiv.org e-Print archive.

[14] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving in DLV. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer, 2000.

[15] T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *International Conference on Logic Programming (ICLP)*, pages 163–177, 2003.

[16] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *International Conference on Database Theory (ICDT)*, pages 207–224. Springer-Verlag, LNCS 2572, 2003.

[17] A. Fuxman and R. Miller. Towards Inconsistency Management in Data Integration Systems. In *IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, 2003.

[18] G. Greco, S. Greco, and E. Zumpano. A Logic Programming Approach to the Integration, Repairing and Querying of Inconsistent Databases. In *International Conference on Logic Programming (ICLP)*, pages 348–364. Springer-Verlag, LNCS 2237, 2001.

[19] G. Greco, S. Greco, and E. Zumpano. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1389–1408, 2003.

[20] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[21] P. C. Kanellakis. Elements of Relational Database Theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1158. Elsevier/MIT Press, 1990.

[22] D. Van Nieuwenborgh and D. Vermeir. Preferred Answer Sets for Ordered Logic Programs. In *European Conference on Logics for Artificial Intelligence (JELIA)*, pages 432–443. Springer-Verlag, LNAI 2424, 2002.

[23] J. Wijsen. Condensed Representation of Database Repairs for Consistent Query Answering. In *International Conference on Database Theory (ICDT)*, pages 378–393. Springer-Verlag, LNCS 2572, 2003.