Efficient Querying of Inconsistent Databases with Binary Integer Programming [*]

Phokion G. Kolaitis
UC Santa Cruz and
IBM Research - Almaden
kolaitis@cs.ucsc.edu

Enela Pema
UC Santa Cruz
epema@cs.ucsc.edu

Wang-Chiew Tan[†]
UC Santa Cruz
tan@cs.ucsc.edu

ABSTRACT

An inconsistent database is a database that violates one or more integrity constraints. A typical approach for answering a query over an inconsistent database is to first clean the inconsistent database by transforming it to a consistent one and then apply the query to the consistent database. An alternative and more principled approach, known as *consistent query answering*, derives the answers to a query over an inconsistent database without changing the database, but by taking into account all possible *repairs* of the database.

In this paper, we study the problem of consistent query answering over inconsistent databases for the class for conjunctive queries under primary key constraints. We develop a system, called EQUIP, that represents a fundamental departure from existing approaches for computing the consistent answers to queries in this class. At the heart of EQUIP is a technique, based on Binary Integer Programming (BIP), that repeatedly searches for repairs to eliminate candidate consistent answers until no further such candidates can be eliminated. We establish rigorously the correctness of the algorithms behind EQUIP and carry out an extensive experimental investigation that validates the effectiveness of our approach. Specifically, EQUIP exhibits good and stable performance on conjunctive queries under primary key constraints, it significantly outperforms existing systems for computing the consistent answers of such queries in the case in which the consistent answers are not first-order rewritable, and it scales well.

1. INTRODUCTION

An *inconsistent* database is a database that violates one or more integrity constraints. Inconsistencies arise frequently under several different circumstances. For example, errors may be unintentionally introduced or the same information may be entered differently in the same database. Inconsistencies may also arise when combining data from different sources, where each source may represent the same information in a different way. There is a large body of work on *data cleaning* aiming to make meaningful sense of an inconsistent database (see [15] for a survey). In data cleaning, the approach taken is to first bring the database to a consistent state by resolving all conflicts that exist in the database. While data cleaning makes it possible to derive *one* consistent state of the database, this process usually relies on statistical and clustering techniques, which entail making decisions on what information to omit, add, or change; quite often, these decisions are of an ad hoc nature.

An alternative, less intrusive, and more principled approach is the framework of *consistent query answering*, introduced in [1]. In contrast to the data cleaning approach, which modifies the database, the inconsistent database is left *as-is*. Instead, inconsistencies are handled at query time by considering all possible repairs of the inconsistent database, where a *repair* of a database I is a database r that is consistent with respect to the integrity constraints, and differs from I in a "minimal" way. A *consistent answer of a query q on I* is a tuple that belongs to the set $\bigcap\{q(r) : r \text{ is a repair of } I\}$. In words, the consistent answers of q on I are the tuples that lie in the intersection of the results of q applied on each repair of I.

There has been a substantial body of research on delineating the boundary between the tractability and intractability of consistent answers, and on efficient techniques for computing them (see [6] for a survey). The parameters in this investigation are the class of queries, the class of integrity constraints, and the types of repairs. Here, we focus on the class of conjunctive queries under primary key constraints and subset repairs, where a *subset repair* of I is a maximal sub-instance of I that satisfies the given integrity constraints. Conjunctive queries, primary keys, and subset repairs constitute the most extensively studied types of, respectively, queries, integrity constraints, and repairs considered in the literature to date.

Earlier systems for computing the consistent answers of conjunctive queries under primary key constraints have been based either on *first-order rewriting* techniques or on heuristics that are developed on top of *logic programming* under *stable sets semantics*.

As the name suggests, first-order rewriting techniques [1, 20, 29] are applicable whenever the consistent answers of a query q can be computed by directly evaluating some other first-order query q' on the inconsistent database at hand. First-order rewritability is appealing because it implies that the consistent answers of q have polynomial-time data complexity and, in fact, can be computed using a database engine. In particular, for the subclass C_{forest} of conjunctive queries under primary key constraints, the system ConQuer, which is developed on top of a relational database management system, was shown to achieve very good performance [19]. However, first-order rewriting techniques can only be applied to a rather limited collection of conjunctive queries under primary key constraints. The reason is that computing the consistent answers of conjunctive queries under primary key constraints is a

[*]Research on this paper was supported by NSF grant IIS-0905276
[†]Part of this work was done while Tan was at IBM Research - Almaden.

coNP-complete problem in data complexity [2]. Moreover, coNP-completeness can arise even for very simple Boolean conjunctive queries with only two atoms under primary key constraints [20, 24].

Heuristics for computing the consistent answers using *logic programming* under *stable model semantics* have high worst-case complexity, but apply to much larger classes of queries and constraints [2, 4, 14, 22, 23, 26]. Indeed, this technique is applicable to arbitrary first-order queries under *universal constraints*. In particular, it is applicable to arbitrary conjunctive queries under primary key constraints, regardless of whether or not their consistent answers are first-order rewritable. However, as we shall show in Section 6, this technique does not scale well to larger databases.

Contributions We propose a different approach that leverages Binary Integer Programming (BIP) to compute the consistent answers to conjunctive queries under primary key constraints.

- We give an explicit polynomial-time reduction from the complement of the consistent answers of a conjunctive query under primary key constraints to the solvability of a binary integer program.
- We present an algorithm that computes the consistent answers of a query using the above binary integer program. Our algorithm repeatedly executes and augments the binary integer program with additional constraints to eliminate candidate consistent answers until no more such candidates can be eliminated.
- We have built a system, called EQUIP, that implements our BIP-based approach over a relational database management system and a BIP solver.
- We have conducted an extensive suite of experiments over both synthetic data and data derived from TPCH to determine the feasibility and effectiveness of EQUIP. Our experimental results show that EQUIP exhibits good performance with reasonable overheads on queries for which computing the consistent answers ranges from being first-order rewritable to coNP-complete. Furthermore, EQUIP performs significantly better than existing systems for queries for which no first-order rewriting exists, and also scales well.

2. RELATED WORK

Computing the consistent answers to conjunctive queries under primary key constraints has been the target of an in-depth investigation within the broader area of consistent query answering. Early intractability results revealed that the data complexity of this problem is coNP-complete [10]; this gave rise to the challenge to identify tractable cases of this problem and also to design useful heuristics for the intractable cases.

Much of the pursuit of tractable cases of consistent query answering has focused on the *first-order rewriting* technique. This technique amounts to taking the original query q, together with the constraints Σ, and constructing a first-order query q' such that the usual evaluation of q' on the inconsistent database I returns exactly the consistent answers to q on I with respect to Σ. In what follows, if such a query q' exists, we say that q is *first-order rewritable*. The main advantage of this approach is the ease of incorporating it into existing database applications, since the query q', being a first-order query, can be efficiently evaluated using a database engine.

The first-order rewriting technique was initially introduced in [1] and further studied by Fuxman et al. [19, 20] and by Wijsen [27, 29]. In particular, Fuxman et al. identified a subclass, called C_{forest}, of self-join free conjunctive queries that are first-order rewritable under primary key constraints, and also built a system, called ConQuer, which implements a rewriting strategy for queries in C_{forest}. In [29], Wijsen studied the class of self-join free acyclic conjunctive queries and characterized the first-order rewritable ones by finding an efficient necessary and sufficient condition for an acyclic conjunctive query under primary key constraints to be first-order rewritable. The two classes of first-order rewritable queries identified by Fuxman et al. and by Wijsen are incomparable. Indeed, on the one hand, C_{forest} contains cyclic first-order rewritable queries (hence these are not covered by Wijsen's results) and, on the other, there are acyclic conjunctive queries that are first-order rewritable, but are not in C_{forest}. We will see such examples in Section 6 of the paper. It remains an open problem to characterize which conjunctive queries are first-order rewritable under primary key constraints.

First-order rewritability is a sufficient, but not necessary, condition for tractability of consistent query answering. There are conjunctive queries, as simple as involving only two different binary relations, for which a first-order rewriting does not exist. Concretely, Wijsen [28] showed that the consistent answers of the conjunctive query $\exists x, y.R_1(x, y) \wedge R_2(y, x)$, where the first attribute of R_1 and R_2 is a key, are polynomial-time computable, but the query is not first-order rewritable. In [24], an effective necessary and sufficient condition was given for a conjunctive query involving only two different relations to have its consistent answers computable in polynomial-time. No such characterization is known for queries involving three or more relations.

A different approach to tractable consistent query answering is the *conflict-hypergraph* technique, introduced by Arenas et al. [3] and further studied by Chomicki et al. in [11, 12]. The conflict-hypergraph is a graphical representation of the inconsistent database in which nodes represent database facts and hyperedges represent minimal sets of facts that together give rise to a violation of the integrity constraints, where the class of integrity constraints can be as broad as *denial constraints*. Chomicki et al. designed a polynomial-time algorithm and built a system, called Hippo, that uses the conflict-hypergraph to compute the consistent answers to projection-free queries that may contain union and difference operators. While the class of constraints supported by Hippo goes well beyond primary key constraints, the restriction to queries without projection limits its applicability. In particular, for the class of conjunctive queries and primary key constraints, this technique does not bring much to the table, as conjunctive queries that are self-join free and projection-free belong to C_{forest}.

In a different direction, *disjunctive logic programming* and *stable model semantics* have been used to compute the consistent answers of arbitrary first-order queries under broad classes of constraints, which include primary key constraints as a special case [2, 4, 14, 22, 23, 26]. For this, disjunctive rules are used to model the process of repairing violations of constraints. These rules form a disjunctive logic program, called the *repair program*, whose stable models are tightly connected with the repairs of the inconsistent database (and in some cases are in one-to-one correspondence with the repairs). For every fixed query, the *query program* is formed by adding a rule on top of the repair program. Query programs can be evaluated using engines, such as DLV [13], for computing the stable models of disjunctive logic programs, a problem known to be Π_2^p-complete. Two systems that have implemented this approach are Infomix [25] and ConsEx [8, 9]. The latter uses the *magic sets* method [5] to eliminate unnecessary rules and generate more compact query programs. Clearly, these systems can be used to compute the consistent answers of conjunctive queries under primary key constraints, which are the object of our study here. However, they can also handle constraints and queries for which the data complexity of consistent query answering is Π_2^p-complete; for example, the data complexity of conjunctive queries under functional and inclusion dependencies is Π_2^p-complete [10]. As mentioned earlier, the data complexity of conjunctive queries under primary key constraints is coNP-complete; our system EQUIP uses a solver for BIP, a problem of matching computational complexity.

Finally, Flesca at al. [17] studied the problem of repairing and querying databases with numerical attributes. To this effect, they used *Mixed Integer Linear Programming* to model repairs based on a minimal number of updates at attribute level, and to extract consistent answers from inconsistent numerical databases. This approach is similar to the BIP approach adopted here, but is focused on numerical databases and on aggregate constraints. In subsequent investigations [16, 18], they used *Integer Linear Programming* to compute the consistent answers of Boolean aggregate queries, as well as the range-consistent answers of `SUM`, `MIN`, and `MAX` queries, in this framework.

3. PRELIMINARIES

Basic Notions A *relational database schema* is a finite collection $\mathbf{R}$ of relation symbols, each with an associated arity. If R is a relation symbol in $\mathbf{R}$ and I is an instance over $\mathbf{R}$, then R^I denotes the interpretation of R on I. Furthermore, if $\mathrm{Attr}(R)$ is the set of all attributes of R, then a *key* of R is a minimal subset X of $\mathrm{Attr}(R)$ for which the functional dependency $X \rightarrow \mathrm{Attr}(R)$ holds; such a functional dependency is called a *key constraint*. We assume that each relation symbol comes with a fixed key.

A *conjunctive query* is a first-order formula built from atomic formulas, conjunctions, and existential quantification. Thus, every conjunctive query is logically equivalent to an expression of the form $q(\mathbf{z}) = \exists \mathbf{w}.R_1(\mathbf{x_1}) \wedge \ldots \wedge R_m(\mathbf{x_m})$, where each $\mathbf{x}_i$ is a tuple of variables and constants, $\mathbf{z}$ and $\mathbf{w}$ are tuples of variables, the variables in $\mathbf{x_1}, \ldots, \mathbf{x_m}$ appear in exactly one of $\mathbf{z}$ and $\mathbf{w}$, and each relation symbol may appear more than once. We will often write conjunctive queries as *rules*; specifically, the rule expressing the preceding conjunctive query is $q(\mathbf{z})\ :-R_1(\mathbf{x_1}), \ldots, R_m(\mathbf{x_m})$. A *Boolean* conjunctive query is a conjunctive query in which all variables are existentially quantified; thus, when a Boolean conjunctive query q is written as a rule, the left-hand side of the rule is $q()$. If a conjunctive query has repeated relation names, we say that it contains a *self-join*. We refer to queries that do not contain self-joins as *self-join free* queries. In what follows, whenever we write a conjunctive query, we underline in each atom variables that appear in the positions of attributes that are part of the key of the relation symbol; such variables are called *key variables*. For example, by writing $\exists x, y.R_1(\underline{x}, y) \wedge R_2(\underline{y}, x)$, we indicate that the first attributes of R_1 and R_2 are, respectively, the keys of R_1 and R_2; furthermore, x is a key variable of the atom $R_1(\underline{x}, y)$, while y is a key variable of the atom $R_2(\underline{y}, x)$. In general, when a conjunctive query is presented in this form, we omit explicitly specifying the schema and the key constraints, since they can be derived from the formulation of the query itself.

A *fact* of an instance I is an expression of the form $R^I(a_1, \ldots, a_n)$ such that $(a_1, \ldots, a_n) \in R^I$. We say that two facts $R^I(a_1, \ldots, a_n)$ and $R^I(b_1, \ldots, b_n)$ *form a conflict* if the tuples $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ witness a violation of the key constraint of R; we also say that these two facts are *key-equal*. A *key-equal group* of facts in a database I is a maximal set of key-equal facts, i.e., it is a set K of facts from I such that every two facts from K are key-equal, and no fact from K is key-equal to some fact from $I \setminus K$.

Database repairs and consistent answers We now give precise definitions of the notions of a subset repair and consistent answers.

DEFINITION 1. Let $\mathbf{R}$ be a relational database schema and Σ a set of primary key constraints over $\mathbf{R}$.

- Let I be an instance. An instance r is a *subset repair* or, simply, a *repair* of I w.r.t. Σ if r is a maximal sub-instance of I that satisfies Σ, i.e., $r \models \Sigma$ and there is no instance r' such that $r' \models \Sigma$ and $r \subset r' \subset I$.
- Let q be a conjunctive query over the schema $\mathbf{R}$.
  - If I is an instance, then a tuple $\mathbf{t}$ is a *consistent answer* of q on I if for every repair r of I w.r.t. Σ, we have that $\mathbf{t} \in q(r)$.

    In the case in which q is Boolean, we say that *true* is the *consistent answer* to q on I if for every repair r of I w.r.t. Σ, we have that $q(r)$ is *true*; otherwise, *false* is the consistent answer of q on I.
  - CERTAINTY(q) is the following decision problem: Given an instance I and a tuple $\mathbf{t}$, is $\mathbf{t}$ a consistent answer of q on I?

    In the case in which q is Boolean, CERTAINTY(q) is the following decision problem: Given an instance I, is *true* the consistent answer to q on I?

It should be pointed out that the decision problem CERTAINTY(q) is about the *data complexity* of consistent query answering, that is to say, the query and the constraints are fixed, hence the complexity of computing the consistent answers depends only on the size of the database instance. It is easy to see that for every fixed conjunctive query q and every set Σ of primary key constraints, CERTAINTY(q) is in coNP. Moreover, there are conjunctive queries and primary key constraints for which CERTAINTY(q) is coNP-complete. For example, as shown in [20], this is the case for the conjunctive query $\exists x, x'y.R_1(\underline{x}, y) \wedge R_2(\underline{x'}, y)$.

Let q be a non-Boolean conjunctive query and let Σ be the set of primary key constraints in q. Since q is a monotone query and since the repairs of a database instance I are sub-instances of I, we have that the consistent answers of q on I form a subset of $q(I)$. Thus, every tuple in $q(I)$ is a candidate for being a consistent answer to q on I w.r.t. Σ. We refer to the tuples in $q(I)$ as the *potential consistent answers* to q on I w.r.t. Σ, or, simply, the *potential answers* to q on I w.r.t. Σ. In the case in which q is Boolean, if $q(I)$ is *false*, then the consistent answer to q is also *false*. If $q(I)$ is *true*, then *true* is the potential answer to q on I. The *consistent part* of I is the the sub-instance of I consisting of all facts of I that are not involved in any conflict. Clearly, the consistent part of a database I is the intersection of all repairs of I. It is also the union of all singleton key-equal groups.

Integer Linear Programming Integer Linear Programs (ILP) are optimization problems of the form $max\ \{\mathbf{c}^T\mathbf{x} \mid A\mathbf{x} \leq \mathbf{b}; x \in \mathbb{Z}^*\}$ (or, in the dual form $min\ \{\mathbf{b}^T\mathbf{y} \mid A^T\mathbf{y} \geq \mathbf{c}; y \in \mathbb{Z}^*\}$), where $\mathbf{b}$ and $\mathbf{c}$ are vectors of integer coefficients, $\mathbf{b}^T$ and $\mathbf{c}^T$ are the transpose of, respectively, $\mathbf{b}$ and $\mathbf{c}$, A is a matrix of integer coefficients and $\mathbf{x}$ (or, $\mathbf{y}$) is a vector of variables, ranging over the set $\mathbb{Z}^*$ of the non-negative integers. The function $\mathbf{c}^T\mathbf{x}$ (or, $\mathbf{b}^T\mathbf{y}$) is called the *objective function*. The system of inequalities $A\mathbf{x} \leq \mathbf{b}$ (or, $A^T\mathbf{y} \geq \mathbf{c}$) are the constraints to the program. Binary Integer Programming (BIP) is the special case of Integer Linear Programming in which the variables must take values from the set $\{0, 1\}$. A *solution* to an integer program is an assignment of non-negative integers to the variables of the program that satisfies the constraints. An *optimal solution* to the integer program is a solution that yields the optimal value of the objective function. If x is the vector of variables in the program, we will use the notation $\hat{x}$ to denote a solution of the integer program, and the notation x^* to denote an optimal solution.

The decision problem underlying Integer Linear Programming is: Given a system $A\mathbf{x} \leq \mathbf{b}$ of linear inequalities with integer coefficients, does it have a solution? Similarly, the decision problem underlying Binary Integer Programming is: Given a system $A\mathbf{x} \leq \mathbf{b}$ of linear inequalities with integer coefficients, does it have a solution consisting of 0's and 1's? It is well known that both these decision problems are NP-complete (see [21]). In what follows, we will use the terms Integer Linear Programming (ILP) to refer to both the optimization problem and the decision problem, and similarly for Binary Integer Programming (BIP).

4. CONSISTENT QUERY ANSWERING VIA BINARY INTEGER PROGRAMMING

From now on, whenever we are working with a conjunctive query q, we will assume that we are given both the conjunctive query and the primary key constraints of the schema of q.

Let q be a conjunctive query. Since CERTAINTY(q) is in coNP and since Binary Integer Programming is NP-complete, there is a polynomial-time reduction from the complement of CERTAINTY(q) to Binary Integer Programming. Here, we will explore explicit natural reductions between these two problems and will develop a technique for computing the consistent answers of conjunctive queries using Binary Integer Programming. The following results will be presented in this section.

- An explicit polynomial-time reduction from the complement of CERTAINTY(q), where q is a Boolean conjunctive query, to Binary Integer Programming. This result is stated as Theorem 1.
- An explicit polynomial-time reduction from the complement of CERTAINTY(q), where q can be a Boolean or a non-Boolean conjunctive query, to Binary Integer Programming. This result is stated as Theorem 2; the technique used in Theorem 2 extends the reduction used in Theorem 1 to non-Boolean queries.
- An algorithm for computing the consistent query answers of a conjunctive query, which uses as a building block the system of constraints in the reduction used in Theorem 2.

Boolean Conjunctive Queries In what follows, we will make use of the notion of a *minimal witness* to a query on a database instance. Let I be a database instance and S a sub-instance of I. We say that S is a *minimal witness* to a Boolean query q if q is *true* on S and for every proper subset S' of S, we have that q is false on S'.

THEOREM 1. *Let q be a fixed Boolean conjunctive query. Given a database instance I over the same schema as q, we construct in polynomial time the following system of linear equalities and inequalities with variables $x_{f_1}, \cdots, x_{f_i}, \cdots, x_{f_n}$, where each variable x_{f_i} is associated with a fact f_i of I.*

System (1):

(a) $\sum_{f_i \in K} x_{f_i} = 1,$ *for K a key-equal group of I.*

(b) $\sum_{f_i \in S} x_{f_i} \leq |S| - 1,$ *for S a minimal witness to q on I, where* $x_{f_i} \in \{0, 1\}$, *for every* $f_i \in I$.

Then the following statements are equivalent:

- *There is a repair r of I in which q is false.*
- *System* (1) *has a solution.*

PROOF. *(Sketch)* Intuitively, the constraints (a) express the fact that from every key-equal group of facts, exactly one fact must appear in a repair. The constraints (b) express the fact that for every minimal witness S to q, not every fact of S should appear in a repair. There is a one-to-one mapping between the repairs in which q is false and the solutions to the set of constraints (a) and (b). Given a repair r of I on which q is false, one can construct a solution $\hat{x}$ by assigning $\hat{x}_{f_i} = 1$ if and only if f_i is a fact in r. In the other direction, given a solution $\hat{x}$ to the constraints (a) and (b), one can construct a repair r of I in which q is false by adding to r precisely the facts f_i such that $\hat{x}_{f_i} = 1$. $\square$

We illustrate Theorem 1 with Example 1 next.

EXAMPLE 1. Let q be the query $q() : -R_1(\underline{x}, y, z), R_2(\underline{x'}, y)$. Let I be the displayed database, where $f_1, \cdots, f_5$ are names used

R_1		**A**	**B**	**C**
	f_1	a	b$_1$	c$_1$
	f_2	a	b$_2$	c$_1$
	f_3	e	b$_1$	c$_2$

R_2		**D**	**E**
	f_4	d	b$_1$
	f_5	d	b$_2$

to identify database facts. For every fact f_i, we introduce a Boolean variable x_{f_i}. Since $\{f_1, f_2\}$ forms a key-equal group of facts, we create the constraint $x_{f_1} + x_{f_2} = 1$. Doing the same for all the key-equal groups, we obtain the equalities (a). The sets of facts that are minimal witnesses to q on I are $\{f_1, f_4\}$, $\{f_2, f_5\}$, and $\{f_3, f_4\}$. From these minimal witnesses, we obtain the inequalities (b).

$$(a) \quad \begin{array}{l} x_{f_1} + x_{f_2} = 1 \\ x_{f_3} = 1 \\ x_{f_4} + x_{f_5} = 1 \end{array} \qquad (b) \quad \begin{array}{l} x_{f_1} + x_{f_4} \leq 1 \\ x_{f_2} + x_{f_5} \leq 1 \\ x_{f_3} + x_{f_4} \leq 1 \end{array}$$

It is easy to check that the consistent answer to q is *false*, because q is false on the repair $r = \{f_1, f_3, f_5\}$. This gives rise to a solution to the system of the constraints (a) and (b) by assigning value 1 to a variable x_i if and only if $f_i \in r$. Thus, we obtain the solution $(x_{f_1}, x_{f_2}, x_{f_3}, x_{f_4}, x_{f_5}) = (1, 0, 1, 0, 1)$. $\square$

The size of System (1) in Theorem 1 is polynomial in the size of the database I (however, the degree of the polynomial depends on the fixed query q). Indeed, there are $|I|$ different variables, that is, as many variables as facts in I. One equality constraint is introduced for every key-equal group. Since every database fact belongs to exactly one key-equal group, the number of constraints in (a) is at most $|I|$. One inequality constraint is introduced for every minimal witness. If q has k atoms, then there are at most $|I|^k$ different minimal witnesses.

Non-Boolean Conjunctive Queries Theorem 1 gives rise to a technique for computing the consistent answers of a Boolean conjunctive query under primary key constraints using a BIP solver. This technique can be extended to non-Boolean queries as follows.

Let q be a conjunctive query of arity k, for some $k \geq 1$. If I is a database instance and $\mathbf{t}$ is a k-tuple with values from the active domain of I, then $\mathbf{t}$ is a consistent answer of q on I if and only if for every repair r of I, we have that $\mathbf{t}$ is in $q(r)$. This is the same as the Boolean query $q[\mathbf{t}]$ being true in every repair r of I, where $q[\mathbf{t}]$ is the query obtained from q by substituting variables from the head of q (i.e., variables that are not existentially quantified) with corresponding constants from $\mathbf{t}$. Thus, for every potential answer $\mathbf{a}$ to q, we can use Theorem 1 to check if $q[\mathbf{t}]$ is true in every repair.

The preceding approach to computing the consistent answers of a non-Boolean query q on some database instance I requires that we solve as many instances of BIP as the number of potential answers of q on I. Hence, if the number of potential answers is large, it is conceivable that this approach may be expensive in practice. For this reason, we explore a different technique for handling non-Boolean queries that avoids constructing and evaluating a binary integer program for every potential answer. We will present this technique in two steps. First, we give a reduction that, given a database instance I, constructs a system of linear equalities and inequalities such that one can reason about all the potential answers to q by exploring the set of solutions to the system. This result, which is stated in Theorem 2, serves as a building block for Algorithm ELIMINATEPOTENTIALANSWERS, which is presented later in this section.

THEOREM 2. *Let q be a fixed conjunctive query. Given a database instance I over the same schema as q, we construct in polynomial time the following system of linear equalities and inequalities with variables $x_{f_1}, \cdots, x_{f_i}, \cdots, x_{f_n}$, where each variable x_{f_i} is associated with a database fact f_i.*

System (2):

(a) $\sum_{f_i \in K} x_{f_i} = 1,$ *for K a key-equal group of I.*

(b) $(\sum_{f_i \in S} x_{f_i}) - u_{\mathbf{a}} \leq |S| - 1,$ *for $\mathbf{a} \in q(I)$ and for S a witness of $q[\mathbf{a}]$ on I, where $x_{f_i}, u_{\mathbf{a}} \in \{0,1\}$, for every $f_i \in I$ and $\mathbf{a} \in q(I)$.*

Then the following statements are equivalent for $\mathbf{a}$, where $\mathbf{a}$ is a potential answer to q on I:

- *There is a repair r of I in which $q[\mathbf{a}]$ is false.*
- *System (2) has a solution $(\hat{x}, \hat{u})$ such that $\hat{u}_{\mathbf{a}} = 0$.*

PROOF. *(Sketch)* If r is a repair of I such that $\mathbf{a}$ is not an answer to q on r, we can construct a solution $(\hat{x}, \hat{u})$ by assigning $\hat{x}_{f_i} = 1$ if and only if f_i is a fact in r, and by assigning $\hat{u}_{\mathbf{a}} = 0$, and $\hat{u}_{\mathbf{b}} = 1$ for every other variable $u_{\mathbf{b}}$, where $\mathbf{b} \neq \mathbf{a}$. In the other direction, given a solution $(\hat{x}, \hat{u})$ such that $\hat{u}_{\mathbf{a}} = 0$, we can construct a repair r to I that does not satisfy $q[\mathbf{a}]$ by adding to r precisely the facts f_i such that $\hat{x}_{f_i} = 1$. $\square$

Observe that for every solution to System (2), there is a database repair that corresponds to the solution, and for every repair, there is a solution to System (2) that corresponds to the repair. Furthermore, it is straightforward to verify that if only the constraints (a) are considered, then the solutions (to the constraints (a)) are in one-to-one correspondence with the repairs of I.

We illustrate Theorem 2 with Example 2.

EXAMPLE 2. Consider the unary query $q(z)$, where $q(z) : -R_1(\underline{x}, y, z), R_2(\underline{x'}, y)$. Let I be the following database:

R_1

	A	B	C
f_1	a	b$_1$	c$_1$
f_2	a	b$_2$	c$_1$
f_3	e	b$_1$	c$_2$
f_4	g	b$_1$	c$_3$
f_5	h	b$_2$	c$_3$

R_2

	D	E
f_6	d	b$_1$
f_7	d	b$_2$

The key-equal groups give rise to equations in (a). The potential answers to q are c_1, c_2 and c_3. To represent each of these potential answers, we use variables u_{c_1}, u_{c_2} and u_{c_3}, respectively. Since $\{f_1, f_6\}$ is a minimal witness to potential answer c_1, we generate the inequality $x_{f_1} + x_{f_6} - u_{c_1} \leq 1$. The set $\{f_2, f_7\}$ is also a minimal witness to c_1. Hence, we generate the inequality $x_{f_2} + x_{f_7} - u_{c_1} \leq 1$. Doing the same for every minimal witness to every potential answer, we create the set of inequalities in (b):

$$(a) \quad \begin{array}{l} x_{f_1} + x_{f_2} = 1 \\ x_{f_3} = 1 \\ x_{f_4} = 1 \\ x_{f_5} = 1 \\ x_{f_6} + x_{f_7} = 1 \end{array} \qquad (b) \quad \begin{array}{l} x_{f_1} + x_{f_6} - u_{c_1} \leq 1 \\ x_{f_2} + x_{f_7} - u_{c_1} \leq 1 \\ x_{f_3} + x_{f_6} - u_{c_2} \leq 1 \\ x_{f_4} + x_{f_6} - u_{c_3} \leq 1 \\ x_{f_5} + x_{f_7} - u_{c_3} \leq 1 \end{array}$$

The vector $(\hat{x}, \hat{u})$ with $\hat{x} = (0,1,1,1,1,1,0)$ and $\hat{u} = (0,1,1)$ is a solution of the system (a) and (b). It is easy to check that the database instance $r = \{f_i \in I \mid \hat{x}_{f_i} = 1\}$ is a repair of I in which c_1 is not an answer to q. Hence, c_1 is not a consistent answer. Similarly, $\hat{x} = (0,1,1,1,1,0,1)$ and $\hat{u} = (1,0,1)$ form a solution of the system. The instance $r = \{f_i \in I \mid \hat{x}_{f_i} = 1\}$ is a repair of I in which c_2 is not an the answer to q. As Theorem 2 states, it is not a coincidence that $\hat{u}_{c_1} = 0$. On the other hand, c_3 is a consistent answer of q. Indeed, since in every solution of the above system it must hold that $\hat{x}_{f_4} = 1$ and $\hat{x}_{f_5} = 1$, and one of $\hat{x}_{f_6}$ or $\hat{x}_{f_7}$ must be assigned value 1, it follows that in order to satisfy both inequalities $x_{f_4} + x_{f_6} - u_{c_3} \leq 1$ and $x_{f_5} + x_{f_7} - u_{c_3} \leq 1$ the variable u_{c_3} must always take value 1. $\square$

The size of the system of constraints is polynomial in the size of the database instance $|I|$. Indeed, the number of variables is equal to the number $|I|$ of facts in I plus the number $|q(I)|$ of potential answers. If the query q has k atoms, then $|q(I)|$ is at most $|I|^k$. The number of equality constraints is at most $|I|$. Finally, for every potential answer there are at most $|I|^k$ inequality constraints.

The system of constraints in Theorem 2 can be viewed as a *compact representation* of the repairs, since, as mentioned earlier, the solutions to the program encapsulate all repairs of the database. The conflict-hypergraph and logic programming techniques, which were discussed in Section 2, also provide compact representations of all repairs. Specifically, in the case of the conflict hypergraph, the repairs are represented by the maximal independent sets, while, in the case of logic programming, the repairs are represented by the stable models of the repair program.

Theorem 2 gives rise to the following technique for computing the consistent query answers to a conjunctive query q: given a database instance I, construct System (2), find all its solutions, and then explore the entire space of these solutions to filter out the potential answers that are not consistent. This technique avoids building a separate system of constraints for every potential answer, unlike the earlier technique based on Theorem 1 . However, the number of solutions of System (2) can be as large as the number of repairs. Thus, it is not a priori obvious how these two techniques would compare in practice. In what follows in this paper, we will demonstrate the advantage that Theorem 2 brings.

Algorithm ELIMINATEPOTENTIALANSWERS In a sense, the solutions of System (2) form a compact representation of all repairs together with information about which potential answers are *false* in each repair. This allows us to explore the space of solutions in an efficient manner, so that we can quickly filter out inconsistent potential answers. It also provides the intuition behind our algorithm ELIMINATEPOTENTIALANSWERS that we will describe next.

ELIMINATEPOTENTIALANSWERS is based on two main observations. First, one can differentiate among the solutions to System (2) according to the number of potential answers that they provide evidence for filtering out. Intuitively, a solution can be thought of as being better than another if it makes it possible to filter out a larger number of potential answers than the second one does. Thus, it is reasonable to examine first the solution that allows to filter out the most potential answers. We can model this idea by building a binary integer program that minimizes the sum of all variables $u_{\mathbf{a_j}}$. Second, some solutions may provide redundant information about potential answers that are not consistent as, in practice, it is common that a potential answer may not be found as an answer in more than one repair. The algorithm ELIMINATEPOTENTIALANSWERS, presented in Figure 1, is an iterative process that in each iteration evaluates a binary integer program, uses the optimal solution to filter out potential answers that are not consistent, incorporates this knowledge into the binary integer program by adjusting the constraints, and re-evaluates the tweaked binary integer program to filter out more potential answers.

In Algorithm ELIMINATEPOTENTIALANSWERS, the optimal solution (x^*, u^*) returned is the one that contains the largest number of 0s in u^*, because the binary integer program minimizes the sum $\sum_{j \in [1..p]} u_{\mathbf{a_j}}$. This allows us to filter out right away many potential answers that are not consistent answers. The reason for adding the new constraints at the end of each iteration is to trivially sat-

```
ALGORITHM ELIMINATEPOTENTIALANSWERS
1. Input: $q$ : conjunctive query
          $I$ : database over the same schema as $q$
          $\mathcal{C}$: the set of constraints constructed from $q, I$ as
            described in Theorem 2.
          $\{\mathbf{a_1}, \cdots, \mathbf{a_p}\}$: the set of potential answers to $q$ on $I$
2. let CONSISTENT be a boolean array with subscripts $1, \ldots, p$
   CONSISTENT$[j]$ represents the element $a_j$ and every entry is
   initialized to true.
3. let $i := 1$
4. let filter:=true
5. let $\mathcal{C}_1 := \mathcal{C}$
6. while (filter=true)
7.     let $P_i = \min\{\sum_{j\in[1..p]} u_{\mathbf{a_j}} |$subject to $\mathcal{C}_i\}$
8.     Evaluate $P_i$ using BIP engine
9.     let $(x^*, u^*)$ be an optimal solution for $P_i$
10.    let $\mathcal{C}_{i+1} := \mathcal{C}_i$
11.    let filter:=false
12.    for $j := 1$ to $p$
13.        if $u^*_{\mathbf{a_j}} = 0$ then
14.            let CONSISTENT[j]:=$false$
15.            Add to system $\mathcal{C}_{i+1}$ the equality $(u_{\mathbf{a_j}} = 1)$
16.            let filter:=true
17.    let $i := i + 1$
18. for $j := 1$ to $p$
19.    if CONSISTENT$[j] = true$
20.        return $\mathbf{a_j}$
```

Figure 1: Algorithm for computing consistent query answers by eliminating potential answers.

isfy the constraints of type (b) related to a potential answer that we have already established that it is not consistent. Each time that we modify the program and re-evaluate it, we filter out more and more potential answers. In theory, the worst case scenario is encountered when we have to modify and re-evaluate the program as many times as there are potential answers. This scenario is very unlikely to encounter in practice, especially when the number of potential answers is large. In Section 6, we show experimentally that even over databases with hundreds of thousands of tuples, two to four iterations will suffice to filter out all the potential answers that are not consistent. Note that if q is a Boolean conjunctive query, then the only potential answer is *true*.

THEOREM 3. *Let q be a conjunctive query and I a database instance. Then, Algorithm* ELIMINATEPOTENTIALANSWERS *computes exactly the consistent query answers to q on I.*

PROOF. The proof uses the following two loop invariants:

1. At the i-th iteration, every optimal solution to P_i is also a solution to the constraints C.
2. At the end of the i-th iteration, if *filter* is *true* then the number of elements in CONSISTENT that are *false* is at least i.

The first loop invariant follows easily from the fact that $\mathcal{C}_i$ contains all the constraints of $\mathcal{C}$. The second loop invariant is proved by induction on i. Assume that at the end of the i-th iteration, the number of *false* elements in CONSISTENT is greater than or equal to i. For every $j \in [1..p]$ such that CONSISTENT$[j]$ is *false*, there must exist a constraint $u_{\mathbf{a_j}} = 1$ in $\mathcal{C}_i$. We will show that at the termination of iteration $i + 1$, if *filter* is *true*, then the number of elements in CONSISTENT that are *false* is greater than or equal to $i + 1$. Since *filter* is *true*, the BIP engine has returned an optimal solution (x^*, u^*) to P_{i+1} such that $u^*_{\mathbf{a_j}} = 0$ for at least some $j \in [1..p]$. Notice that it is not possible that at some previous iteration, CONSISTENT$[j]$ has been assigned *false*. If that were the case, then the constraint $u_{\mathbf{a_j}} = 1$ would be in $\mathcal{C}_{i+1}$, and (x^*, u^*) would not be a solution of $\mathcal{C}_{i+1}$. Therefore, at iteration $i + 1$, at least one element in CONSISTENT that has value *true* is changed to *false*.

We show that the algorithm always terminates. The second loop invariant implies in a straightforward manner that the algorithm terminates in at most p iterations. Next, we show that for any $m \in [1..p]$, a potential answer $\mathbf{a_m}$ is a consistent answer if and only if CONSISTENT$[m] = 1$ at termination. In one direction, if $\mathbf{a_m}$ is a consistent answer, then Theorem 2 implies that for every solution $(\hat{x}, \hat{u})$ to the constraints $\mathcal{C}$, it always holds that $\hat{u}_{\mathbf{a_m}} = 1$. Since for every $i \in [1..p]$, every optimal solution to P_i is also a solution to $\mathcal{C}$, we have that the algorithm will never execute line 14 for $j = m$. Hence, the value of CONSISTENT$[m]$ will always remain *true*.

In the other direction, if CONSISTENT$[m]$ is *true* after the algorithm has terminated, then assume towards a contradiction that $\mathbf{a_m}$ is not a consistent answer. If the algorithm terminates at the i-th iteration, then every solution to P_i is such that it assigns value 1 to every variable $u_{\mathbf{a_j}}$, for $j \in [1..p]$. So, the minimal value that the objective function of P_i can take is p. If $\mathbf{a_m}$ were not a consistent answer, we know from Theorem 2 that there must be a solution $(\hat{x}, \hat{u})$ to $\mathcal{C}$ such that $\hat{u}_{\mathbf{a_m}} = 0$. We will reach a contradiction by showing that we can construct from $(\hat{x}, \hat{u})$ a solution (x^*, u^*) to P_i such that $\sum_{j\in[1..p]} u^*_{\mathbf{a_j}} < p$. The vectors $\hat{x}, \hat{u}$ are defined as follows: $x^* = \hat{x}$, $u^*_{\mathbf{a_m}} = \hat{u}_{\mathbf{a_m}}$ and $u^*_{\mathbf{a_j}} = 1$ for $j \neq m$. Because the equality constraints (a) in $\mathcal{C}_i$ and in $\mathcal{C}$ are the same, we have that x^* will satisfy the constraints (a) in $\mathcal{C}_i$. For $j \neq m$ and since $u^*_{\mathbf{a_j}} = 1$, all inequalities in (b) that involve $u^*_{\mathbf{a_j}}$, where $j \neq t$, are trivially satisfied. In addition, the left-hand-side of every inequality in the constraints (b) that involves $u^*_{\mathbf{a_m}}$ will evaluate to the same value under $(\hat{x}, \hat{u})$ and (x^*, u^*) (this follows directly from the fact that $x^* = \hat{x}$ and $u^*_{\mathbf{a_m}} = \hat{u}_{\mathbf{a_m}}$). Finally, all equality constraint that may have been added to $\mathcal{C}$ during previous iterations are satisfied since $u^*_{\mathbf{a_j}} = 1$ for $j \neq m$, and the equality $u^*_{\mathbf{a_m}} = 1$ cannot be in $\mathcal{C}_i$. Now, it is clear that (x^*, u^*) is a solution to P_i such that $\sum_{j\in[1..p]} u^*_{\mathbf{a_j}} = p - 1$. This contradicts the assumption that all solutions to P_i yield minimal value p of the objective function. □

5. IMPLEMENTATION

We have developed a system, called EQUIP, for computing the consistent query answers to conjunctive queries under primary key constraints. EQUIP executes in three phases (see Figure 2), corresponding to the modules: 1) database pre-processor, 2) constraints builder, and 3) consistent query answers (CQA) evaluator.

- PHASE 1: Compute answers of the query from the consistent part of the database and extract database facts that may be relevant for computing the additional answers to the query.
- PHASE 2: Construct the set of constraints based on Theorem 2 with the set of database facts retrieved in PHASE 1.
- PHASE 3: Run Algorithm ELIMINATEPOTENTIALANSWERS.

PHASE 1 The database pre-processor performs two main tasks in phase 1 as an attempt to optimize subsequent phases.

First, it retrieves answers of q that are obtained from the consistent part of the database. Clearly, the result of q over the consistent part of the database is a subset of the consistent query answers of q on I. The set of all consistent query answers that are obtained from

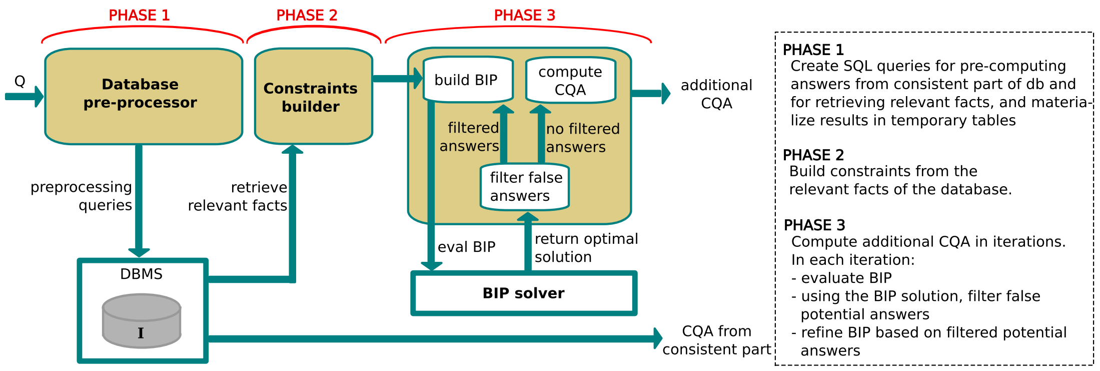
PHASE 1
PHASE 2
PHASE 3
Q
Database pre-processor
Constraints builder
build BIP
compute CQA
filtered answers
no filtered answers
filter false answers
additional CQA
preprocessing queries
retrieve relevant facts
DBMS
I
eval BIP
return optimal solution
BIP solver
CQA from consistent part
PHASE 1
Create SQL queries for pre-computing answers from consistent part of db and for retrieving relevant facts, and materialize results in temporary tables
PHASE 2
Build constraints from the relevant facts of the database.
PHASE 3
Compute additional CQA in iterations.
In each iteration:
- evaluate BIP
- using the BIP solution, filter false potential answers
- refine BIP based on filtered potential answers

Figure 2: Architecture of EQUIP

the consistent part of the database is denoted as ANS_FROM_CON in Figure 3. This set ANS_FROM_CON is immediately returned as part of the consistent answers of q on I (see bottom of Figure 2).

After this, it extracts the set of facts from I that may be *relevant* for computing additional consistent answers of q on I. Since all additional consistent answers belong to ($q(I)$ - ANS_FROM_CON), every fact that shares a key with a fact of a minimal witness to a tuple in ($q(I)$ - ANS_FROM_CON) is a *relevant fact*. In other words, for every tuple $a \in (q(I)$ - ANS_FROM_CON), every fact in a minimal witness S of a is a relevant fact. Furthermore, every fact that shares a key with some tuple $s \in S$ is also a relevant fact. The witnesses and relevant facts are computed as WITNESSES and RELEVANT_R$_i$ as shown in the third and, respectively, fourth steps of Figure 3. The relations KEYS_R$_i$, ANS_FROM_CON, WITNESSES, and RELEVANT_R$_i$ are computed by SQL queries that are executed over the instance I, which is stored in a database. We store the resulting relations in temporary tables (i.e., tables that exist only within the session). The relation KEYS_R$_i$, which is used to compute ANS_FROM_CON, contains all tuples in R_i whose key-equal group has size more than 1.

EXAMPLE 3. Let $q(z) : -R_1(\underline{x}, y, z), R_2(\underline{x'}, y, w)$ be a query over the schema $\{R_1(\underline{A_1}, B_1, C_1), R_2(\underline{A_2}, B_2, C_2)\}$. The following views will be declared in PHASE 1:

KEYS_R$_1(A_1)$: select A$_1$ from R$_1$ group by A$_1$ having count(*)>1

KEYS_R$_2(A_2)$: select A$_2$ from R$_2$ group by A$_2$ having count(*)>1

ANS_FROM_CON(C_1):
```
  select R1.C1
  from R1 inner join R2 on R1.B1=R2.B2
  where R1.A1 not in (select * from KEYS_R1) and
        R2.A2 not in (select * from KEYS_R2)
```

WITNESSES$(A_1, B_1, C_1, A_2, B_2, C_2)$:
```
  select A1,B1,C1,A2,B2,C2
  from R1 inner join R2 on R1.B1=R2.B2
  where R1.C1 not in (select * from ANS_FROM_CON)
```

RELEVANT_R$_1(A_1)$: select * from R$_1$ inner join WITNESSES on R$_1$.A$_1$=WITNESSES.A$_1$

RELEVANT_R$_(A_2)$: select * from R$_2$ inner join WITNESSES on R$_2$.A$_2$=WITNESSES.A$_2$

PHASE 2 This phase uses the temporary tables created in PHASE 1 to build the set of constraints, as described in Theorem 2. Every tuple in RELEVANT_R$_i$ is represented by a variable in the integer program. From every group of facts in RELEVANT_R$_i$ that have the same values in the positions of the key attributes of R$_i$, we add a constraint of type (a) as described in Theorem 2. Every tuple in WITNESSES represents a minimal witness to a potential answer to the query. So, from every tuple in WITNESSES we construct a constraint of the type (b) as described in Theorem 2.

```
PHASE 1: DATABASE PRE-PROCESSING
Input: $\mathbf{R}$ : Schema with relation names $\{R_1, \cdots, R_i, \cdots, R_l\}$
    $q(\mathbf{z}) : -R_{p_1}(\underline{\mathbf{x_1}}, \mathbf{y_1}), \cdots, R_{p_j}(\underline{\mathbf{x_j}}, \mathbf{y_j}), \cdots, R_{p_k}(\underline{\mathbf{x_k}}, \mathbf{y_k})$
      for $j \in [1..k]$ and $1 \le p_j \le l$
    $I$ : database over $\mathbf{R}$
for all $R_i, 1 \le i \le l$
    create view KEYS_R$_i$ that contains all tuples $\mathbf{d}$ s.t.
    there exists more than one fact of the form R$_i$($\mathbf{d}$,_) in $I$

create view ANS_FROM_CON that contains all tuples $\mathbf{t}$ s.t.
    - $\mathbf{t} \in q(I)$, and
    - there exists a minimal witness $S$ for $q(\mathbf{t})$, and s.t.
      no fact $R_i(\mathbf{d}, \_) \in S$ has its key-value $\mathbf{d}$ in KEYS_R$_i$

create view WITNESSES that contains all tuples of the form
($\mathbf{t_{p_1}}, \cdots, \mathbf{t_{p_j}}, \cdots, \mathbf{t_{p_k}}$) s.t.
    - $R_{p_j}(\mathbf{t_{p_j}}) \in I$ for $1 \le j \le k$, and
    - the set S=$\{R_{p_j}(\mathbf{t_{p_j}})\ :\ 1 \le j \le k\}$ forms a minimal witness
        for $q(\mathbf{a})$, where $\mathbf{a}$ is a potential answer that is not in
ANS_FROM_CON

for all $R_i, 1 \le i \le l$
    create view RELEVANT_R$_i$ that contains all tuples $\mathbf{t}$ s.t.
      - $R_i(t) \in I$, and
      - there exists an atom $R_{p_j}(\underline{\mathbf{x_j}}, \mathbf{y_j})$ in $q$ such that $p_j = i$ and
exists a tuple ($\mathbf{t_{p_1}}, \cdots, \mathbf{t_{p_j}}, \cdots, \mathbf{t_{p_k}}$) in WITNESSES
        s.t. $R_{p_j}(\mathbf{t_{p_j}})$ and $R_i(\mathbf{t})$ are key-equal
```

Figure 3: Description of PHASE 1

PHASE 3 In this phase, the additional potential answers are first retrieved by simply taking a projection over the view WITNESSES on the attributes that appear in the head of the query q. With this input, the set of constraints built in PHASE 2, and the database instance, we run Algorithm ELIMINATEPOTENTIALANSWERS next. In the first iteration, we start with a binary integer program whose objective function is the sum of all variables $u_{\mathbf{a_j}}$, where each $u_{\mathbf{a_j}}$ represents a distinct potential answer $\mathbf{a_j}$, and whose constraints are those from PHASE 2. In every iteration, the program is evaluated using a BIP solver. The first optimal solution that is returned by the optimizer is used to filter out potential answers that are false. In each iteration, the integer program is augmented with more constraints. As the constraints we add make some of the existing constraints of type (b) (see Theorem 2) trivially satisfiable, this results in a program (in the next iteration) that is simpler to evaluate than the one evaluated in the current iteration.

6. EXPERIMENTAL EVALUATION

We have conducted an extensive set of experiments with EQUIP. Our goal is to analyze the performance of EQUIP on conjunctive queries whose consistent answers have varying complexity and on databases of varying size and of varying degree of inconsistency (i.e., the percentage of tuples involved in conflicts). We also seek to understand how EQUIP compares with earlier consistent query answering systems, such as ConQuer and ConsEx.

Our experimental results show that EQUIP incurs reasonable overheads, even when the size of the database is scaled up to millions of tuples for both synthetic data and data derived from TPCH. Our results also demonstrate that EQUIP is, by far, the best available system for computing the consistent query answers of conjunctive queries that are not first-order rewritable, as well as for conjunctive queries that are first-order rewritable, but cannot be handled by ConQuer. In particular, EQUIP outperforms ConsEx on coNP-hard queries by a significant margin, especially on larger databases.

6.1 Experimental Setting

Our experiments have been carried out on a machine running on a Dual Intel Xeon 3.4GHz Linux workstation with 4GB of RAM. We use the 64 bit Ubuntu v11.04, DB2 Express C v10.1 as the underlying DBMS, and IBM's ILOG CPLEX v12.3 for solving the binary integer programs. Our system is implemented in Java v1.6.

No established benchmark for consistent query answering exists. Moreover, as noted in [19], conjunctive queries obtained from TPCH queries after removing aggregates, grouping, and subqueries are first-order rewritable and, in fact, most of them are in the class C_{forest}, hence their consistent answers can be computed using the ConQuer system. We have conducted experiments using such queries and data derived from TPCH to compare EQUIP with ConQuer. However, since we are interested in understanding the performance of EQUIP on conjunctive queries whose consistent answers are of varying complexity (from first-order rewritable to coNP-complete), we compiled a list of queries of varying complexity and generated synthetic inconsistent databases for our experiments. We now discuss this experimental setting in some detail.

Benchmark queries Table 1 contains the list of queries we compiled. The queries vary according to the number of atoms, the number of free variables, and the computational complexity of their consistent answers, which can be: first-order rewritable; in PTIME but not first-order rewritable; coNP-complete. Queries Q_1 to Q_{14} are shown not to be first-order rewritable using the results in [29]. For the two-atom queries $Q_1, Q_2, Q_3, Q_8, Q_9, Q_{10}$, their complexity (coNP-hard or PTIME) can be immediately derived from the results in [24]. The coNP-hardness of the three-atom queries Q_4 to Q_7 can be shown via a simple reduction from $\text{CERTAINTY}(q)$, where q is the Boolean query $q() : -R(\underline{x}, y), S(\underline{x'}, y)$. Queries Q_{11} to Q_{14} can be shown to be in PTIME by extending the *conflict-join graph* algorithm introduced in [24] for the queries with two atoms. The first-order rewritability of queries Q_{15}-Q_{21} can be shown using the results in [29]. Moreover, queries Q_{15} to Q_{18} are also in the class C_{forest} [20], which provides another proof that they are first-order rewritable.

Database generation In our experiments with the above queries, we use synthetic databases, which we generate in two steps: (a) generate a consistent database; and (b) generate an inconsistent

Complexity: coNP-complete

$Q_1() : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w)$
$Q_2(z) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w)$
$Q_3(z, w) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w)$
$Q_4() : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, y), R_7(\underline{y}, u, d)$
$Q_5(z) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, y), R_7(\underline{y}, u, d)$
$Q_6(z, w) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w), R_7(\underline{y}, u, d)$
$Q_7(z, w, d) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w), R_7(\underline{y}, u, d)$

Complexity: PTIME, not first-order rewritable

$Q_8() : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w)$
$Q_9(z) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w)$
$Q_{10}(z, w) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w)$
$Q_{11}() : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w), R_7(\underline{y}, u, d)$
$Q_{12}(z) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w), R_7(\underline{y}, u, d)$
$Q_{13}(z, w) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w), R_7(\underline{y}, u, d)$
$Q_{14}(z, w, d) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w), R_7(\underline{y}, u, d)$

Complexity: First-order rewritable

$Q_{15}(z) : -R_1(\underline{x}, y, z), R_2(\underline{y}, v, w)$
$Q_{16}(z, w) : -R_1(\underline{x}, y, z), R_2(\underline{y}, v, w)$
$Q_{17}(z) : -R_1(\underline{x}, y, z), R_2(\underline{y}, v), R_7(\underline{v}, u, d)$
$Q_{18}(z, w) : -R_1(\underline{x}, y, z), R_2(\underline{y}, v), R_7(\underline{v}, u, d)$
$Q_{19}(z) : -R_1(\underline{x}, y, z), R_8(\underline{y, v}, w)$
$Q_{20}(z) : -R_5(\underline{x}, y, z), R_6(\underline{x'}, y, w), R_9(\underline{x}, y, d)$
$Q_{21}(z) : -R_3(\underline{x}, y, z), R_4(\underline{y}, x, w), R_{10}(\underline{x}, y, d)$

Table 1: Benchmark queries used in our experiments

database from the consistent database by inserting tuples that would violate some key constraints. Instead of using available database generators, such as datagen in TPCH for (a), we have generated our own databases for (a) because it is difficult to express over the TPCH schema a variety of meaningful queries that are not first-order rewritable. As part of future work, we plan to leverage more sophisticated generators for (a), such as *QAGen* [7], that would allow one to generate (consistent) databases over a given schema and with varying data distributions.

a) Generation of the consistent databases Each relation in the generated consistent database has the same number (denoted as *r_size*) of facts. The values of the non-key attributes are generated so that for every two atoms R_i, R_j that share variables in any of the queries, approximately 25% of the facts in R_i join with some fact in R_j, and vice-versa. The third attribute in all of the ternary relations, which is sometimes projected out and never used as a join attribute in Table 1, takes values from a uniform distribution in the range [1, *r_size*/10]. Hence, in each relation, there are approximately *r_size*/10 distinct values in the third attribute, each value appearing approximately 10 times. The choice of the distributions is made with the purpose to simulate reasonably high selectivities of the joins and large number of potential answers.

b) Generation of the inconsistent databases From the consistent database, an inconsistent database is obtained by adding, for each relation, tuples that violate some key constraints. The inconsistency generator takes as input a consistent relation and two additional parameters: (a) *nr_conflicts*: the total number of violations to be added to the consistent relation; and (b) *c_size*: the size of each key-equal group. The inconsistent version of a consistent relation r is obtained by repeatedly adding tuples that violate some key value. Specifically, a key value from the key values of the tuples in r, where the number of violations for the key value is less than *c_size* - 1, is first selected. Subsequently, additional distinct tuples with the same key value are added, so that there is a total of *c_size* distinct tuples with the same key value. The non-key attributes of these newly generated tuples are obtained by using the non-key attributes of some randomly selected tuples in r. The purpose of reusing the non-key attributes is to preserve the existing data distributions as we augment r with new tuples. This process is repeated until a total of *nr_conflicts* is obtained. In fact, the inconsistency generator that we have just described is similar to the one used in ConQuer [20], except that we have added the parameter *c_size* to control the sizes of the key-equal groups.

Other Experimental Parameters Unless otherwise stated, our experiments use databases with 10% conflicts in which each key-equal group contains two facts. The *evaluation time* of a consistent query answering system is the time between receiving the query to the time it takes for the system to return the last consistent answer to the query. We always generate five random databases with the same parameters, i.e., with the same size, the same degree of inconsistency, and the same number of facts per key-equal group. For each database, we run each query five times and take the average of the last three runs, and finally, we take the averages of the query evaluation times over the five databases.

One of the CPLEX parameters that can be varied is the *relative GAP parameter*. This parameter is used to specify the level of "tolerance" on the optimality of the solution that is found. The GAP parameter can be set to allow the optimizer to return a less-than-optimal, but "good enough" solution. In other words, the solution that is returned may not be optimal, but will have an estimated gap from the optimal solution that is within the tolerance limit given by the GAP parameter. Having a large GAP parameter usually has the effect of allowing the optimizer to find a solution much earlier. As the correctness of Algorithm 1 is not dependent on obtaining an optimal solution, we have set the GAP parameter to 0.1. In this way, CPLEX may avoid the long running times that may otherwise be incurred by searching for an optimal solution.

6.2 Experimental Results and Comparisons

Our first goal was to determine how the EQUIP system compares against existing systems that are capable of computing the consistent answers of conjunctive queries that are not first-order rewritable. The only other systems that have this capability are the ones that rely on the logic programming technique, such as Infomix and ConsEx. Since ConsEx is the latest system in this category and also has the feature that it implements an important optimization based on *magic sets*, we have compared EQUIP against ConsEx.

Comparison with ConsEx Our findings, depicted in Figure 4, show that EQUIP outperforms ConsEx by a significant margin, especially on larger datasets and even on small queries whose consistent answers are coNP-hard (i.e., queries Q_1 and Q_2)), as well as on the small queries whose consistent answers are in PTIME, but not first-order rewritable (i.e., queries Q_8 and Q_9).

Our second goal was to investigate the total evaluation time and the overhead of EQUIP on conjunctive queries whose consistent answers have varying complexity.

Evaluation time for coNP-hard queries In Figure 5(a), we show the total evaluation time in seconds, using EQUIP, of each query $Q_1 - Q_7$ as the size of the inconsistent database is varied. In Figure 5(b), we show the overhead of computing the consistent answers, relative to the time for evaluating the query over the inconsistent database (i.e., the time for evaluating the potential answers).

Figure 5(a) shows a sub-linear behavior of the evaluation of consistent query answers, as we increase the database size. Note that the overhead for computing consistent answers is rather constant and no more than 5 times the time for usual evaluation. In particular, the Boolean queries Q_1 and Q_4 incur noticeably less overhead than the rest of the queries. The reason is that these queries happen to evaluate to true over the consistent part of the constructed databases, and hence, since there are no additional potential answers to check, no binary integer program is constructed and solved.

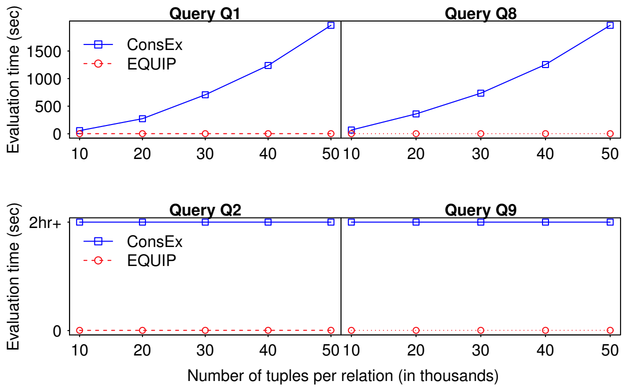

Figure 4: Comparison of EQUIP with ConsEx.

Evaluation time for PTIME, not first-order rewritable queries In Figure 6, we show the performance of EQUIP on conjunctive queries whose consistent answers are in PTIME, but are not first-order rewritable. Since EQUIP is based on Binary Integer Programming, which is an NP-complete problem, we do not expect EQUIP to perform better on such queries as compared to queries that are coNP-hard. Indeed, Figure 5 and Figure 6 show that the performance of EQUIP on the tractable queries Q_8 to Q_{14} is comparable to the performance of EQUIP on the intractable queries Q_1 to Q_7.

Evaluation time for first-order rewritable queries Next, we evaluate EQUIP on first-order rewritable queries. As Figure 7 shows, the performance of EQUIP on such queries is comparable to that on queries in the other classes (see Figure 5 and Figure 6). We also compare EQUIP with ConQuer on queries Q_{15}-Q_{18}, since these queries are in C_{forest}, hence their consistent answers can be computed using ConQuer. The results of this comparison are shown in Figure 8. ConQuer performs better than EQUIP on each of the queries Q_{15} to Q_{18}. This is not surprising, because EQUIP is agnostic to the complexity of the query and "blindly" reduces queries into binary integer programs to be solved by CPLEX. This reduction is exponential in the size of the query and instance, even though these queries are first-order rewritable. These findings suggest that ConQuer is preferred whenever the query is in C_{forest}.

Experiments with data and queries derived from the TPCH benchmark We used the data generator of the TPCH benchmark to generate a consistent database of size 1GB. From this database we created an inconsistent database with 10% conflicts, where each key equal group has size 2. The experiments use queries derived from existing TPCH queries Q_2, Q_3, Q_4, Q_{10}, Q_{11}, Q_{20}, Q_{21} by removing aggregates, grouping, and sub-queries. Due to space limitations, we have omitted spelling out these queries. Also, to avoid naming conflicts with our benchmark queries in Table 1, we shall henceforth denote these queries as Q'_2, Q'_3, Q'_4, Q'_{10}, Q'_{11}, Q'_{20}, and Q'_{21}.

Figure 9 shows the comparison between EQUIP and ConQuer on the more expensive queries Q'_3, Q'_{10}, Q'_{21}, and the less expensive queries Q'_2, Q'_4, Q'_{11}, Q'_{20}. Due to a bug, ConQuer could not evaluate query Q'_{21}. The overheads incurred by EQUIP on experiments with data derived from TPCH are approximately in the same range as the overheads we observed when evaluating the system with our own benchmark queries, and we omit the graphs here. As expected, ConQuer performs better than EQUIP, about twice as fast as EQUIP. These results reinforce our earlier findings for first-order rewritable synthetic queries and synthetic data.

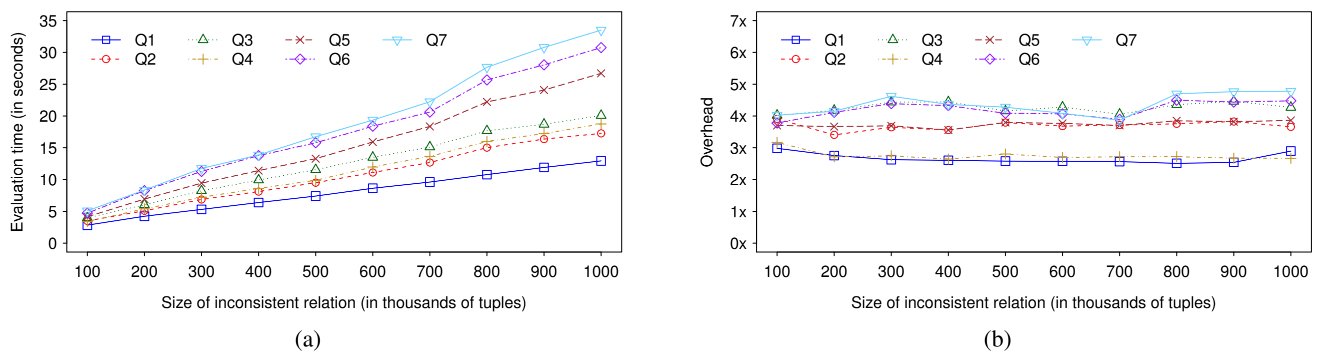

Figure 5: Evaluation time and overhead of EQUIP for computing consistent answers of coNP-hard queries Q_1-Q_7.

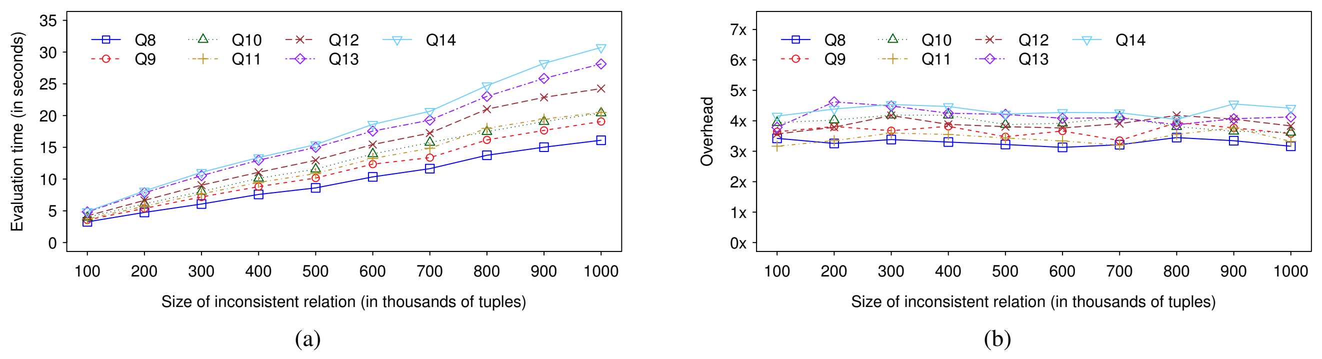

Figure 6: Evaluation and overhead of EQUIP for computing consistent answers of P, but not-first-order rewritable queries Q_8-Q_{14}.

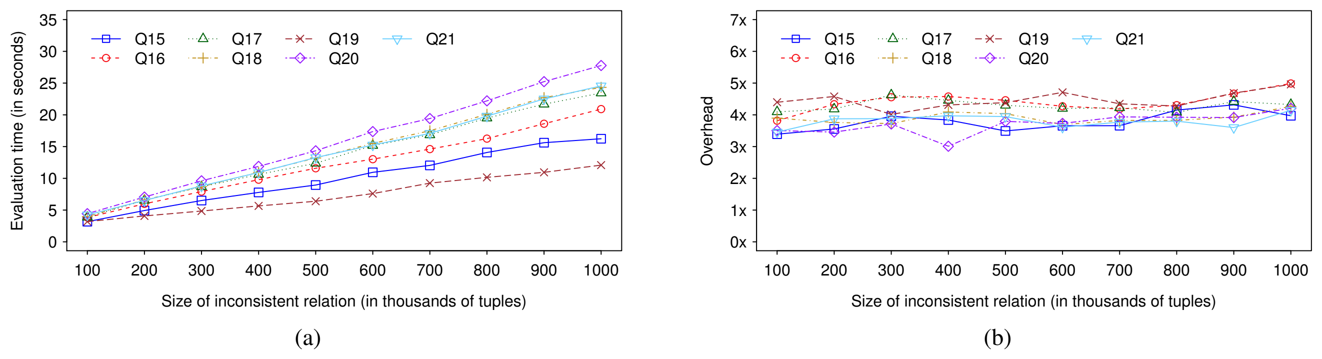

Figure 7: Evaluation and overhead of EQUIP for computing consistent answers of first-order rewritable queries Q_{15}-Q_{21}.

6.3 Further Experimental Results

To gain deeper insights to the performance of EQUIP, we took several other measurements concerning the behavior of EQUIP on the queries in Table 1. We describe our findings next.

Evaluation time per phase Figure 10 shows the evaluation times for queries $Q_1 - Q_{21}$, split into the three phases: PHASE 1, PHASE 2, and PHASE 3. Not surprisingly, PHASE 1, the pre-processing step executed over DB2, dominates the overall evaluation time. The results also show that the construction and evaluation times of the integer program takes more time for non-Boolean queries. Moreover, by looking at Q_6, Q_7, Q_{13}, Q_{14}, we see that PHASE 2 and PHASE 3 take more time for queries that have more variables in the head of the query. In contrast, for the Boolean queries, the BIP solver is not even invoked.

Pre-computation from the consistent part of the database Since we have approximately 10% of the database involved in conflicts, it is natural to expect that a significant portion of the consistent answers can be computed from the consistent part of the database. Hence, we expect that the optimization we have implemented in PHASE 1 for computing some of the consistent answers from the consistent part of the database to play a role in reducing the overall evaluation time. Figure 11 shows the evaluation times of EQUIP, where part of the consistent answers are computed from the consistent part of the database and additional answers are computed using Algorithm ELIMINATEPOTENTIALANSWERS, versus the evaluation time when all of the consistent answers are computed using Algorithm ELIMINATEPOTENTIALANSWERS. Our experiments show that the latter technique (without optimization), on 12 of the queries it is between 1.1 and 1.5 times slower; on 6 of the queries

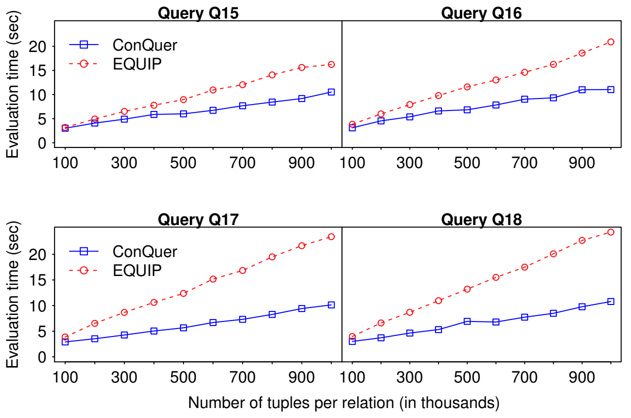

Figure 8: Comparison of EQUIP with ConQuer on queries Q_{15} - Q_{18}, over the database with 1 million tuples/relation.

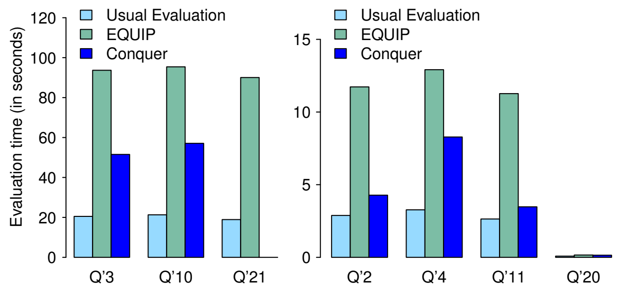

Figure 9: Evaluation times of the simplified TPCH queries over "as-is" query evaluation, EQUIP, and ConQuer.

it is between 1.6 and 2 times slower; on 2 queries it is over 2 times slower; and only on one query it is slightly faster.

Sizes of the binary integer programs In Table 2, we tabulate the sizes of the binary integer programs that were constructed from a database with 1 million tuples per relation. As this table indicates, for some of the queries, large binary integer programs containing over 100,000 variables and constraints are solved. Observe that the number of variables is much smaller than the number of tuples. The reason is that the binary integer programs are built only from facts relevant to computing the additional consistent query answers. Recall that PHASE 1 retrieves only the relevant facts. The Boolean queries Q_1, Q_4, Q_8 and Q_{11} evaluate to true over the consistent part of the databases used for these experiments; hence, there are no relevant facts, and no integer program is constructed. So far, we have not run into the potential limitation that the binary integer program generated may be too large to fit into main memory. This is true even for experiments conducted with data derived from TPCH. In general, instead of building a single program for checking all the potential answers at once, we can, in fact, partition the set of potential answers and construct a "smaller" binary integer program for each bucket. As part of future work, we plan to further investigate this technique.

Varying the degree of inconsistency Figure 12 shows the effect of varying the degree of inconsistency $(10\%, 15\%, 20\%)$ with a database where there is 1 million tuples per relation. Our results indicate that in the worst case, the evaluation time increases linearly with the degree of inconsistency. This is not unexpected as EQUIP has to manage a larger number of relevant tuples as the degree of inconsistency increases. Not surprisingly, the degree of inconsistency affects the percentage of consistent answers out of all the potential answers. Specifically, when the degree of inconsistency is 10%, the percentage of consistent answers, depending on the query, ranges from 80% (for Q_7, Q_{14}) to 99% (for Q_{15}). When the degree of inconsistency is increased to 20%, the percentage of consistent answers ranges from 63% to 97%.

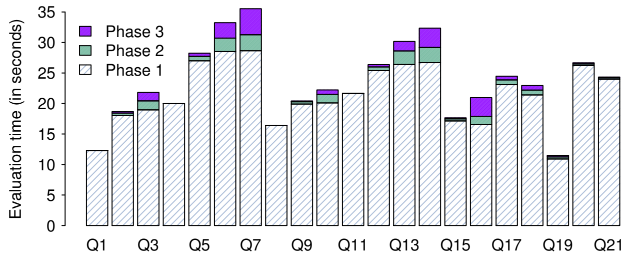

Figure 10: Evaluation time for PHASE 1, PHASE 2, and PHASE 3 of EQUIP over a database with 1 million tuples/relation.

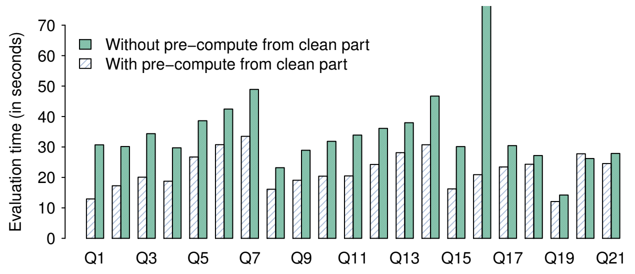

Figure 11: Performance of EQUIP with and without pre-computing the consistent answers from the consistent part of the database, over a database with 1 million tuples/relation.

Query	Nr. variables	Nr. constraints	Query	Nr. variables	Nr. constraints
Q_1, Q_4, Q_8, Q_{11}	0	0	Q_{13}	148K	126K
Q_2	13K	12K	Q_{14}	170K	137K
Q_3	89K	71K	Q_{15}	9K	8K
Q_5	26K	26K	Q_{16}	109K	86K
Q_6	141K	124K	Q_{17}	31K	28K
Q_7	160K	138K	Q_{18}	53K	45K
Q_9	11K	9K	Q_{19}	20K	18K
Q_{10}	86K	64K	Q_{20}	19K	17K
Q_{12}	22K	120K	Q_{21}	16K	17K

Table 2: Size of BIP program (DB has 1 million tuples/relation).

The use of indices We have found out that the majority of the time for computing the consistent query answers goes to PHASE 1. Since PHASE 1 consists of evaluating SQL queries on top of the underlying DBMS, it is natural to consider making use of DBMS features, such as indices. For this reason, we also conducted experiments in which we added a number of indices. For every relation, we have created a clustered index on the key attributes. Figure 13 shows the evaluation of EQUIP in the presence of indices. The improved performance from using indices is clear, as for all the queries the evaluation time has decreased significantly. One may also be able to obtain better performance by using a different configuration of indices, or by a better tuning of DBMS parameters.

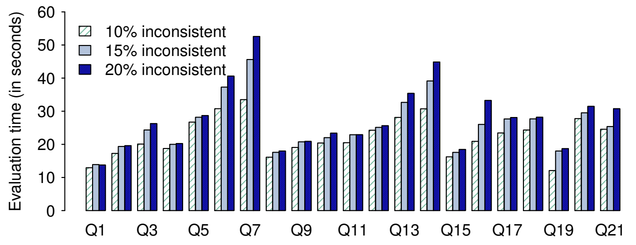

Figure 12: Evaluation time of EQUIP over databases with 1 million tuples/relation and different degrees of inconsistency.

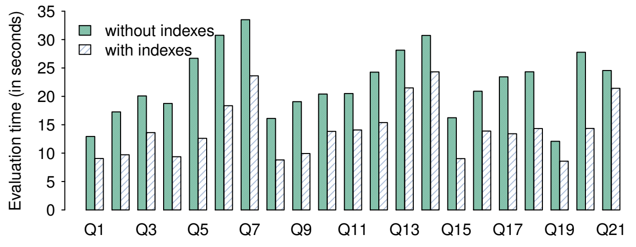

Figure 13: Performance of EQUIP in the presence of indices, over a database with 1 million tuples/relation.

7. CONCLUDING REMARKS

We developed EQUIP, a new system for computing the consistent answers of conjunctive queries under primary key constraints. The main technique behind EQUIP is the reduction of the problem of computing the consistent answers of conjunctive queries under primary key constraints to binary integer programming, and the systematic use of efficient integer programming solvers, such as CPLEX. Our extensive experimental evaluation suggests that EQUIP is promising in that it consistently exhibits good performance even on relatively large databases. EQUIP is also, by far, the best available system for evaluating the consistent query answers of queries that are coNP-hard and of queries that are in PTIME but not first-order rewritable, as well as queries that are first-order rewritable, but not in the class C_{forest}. For C_{forest} queries, ConQuer demonstrates superior performance, but is of limited applicability.

For future work, we plan to investigate extensions of our technique to unions of conjunctive queries and broader classes of constraints, such as functional dependencies and foreign key constraints.

Finally, our results suggest that an "optimal" system for consistent query answering is likely to rely not on a single technique, but, rather, on a portfolio of techniques. Such a system will first attempt to determine the computational complexity of the consistent answers of the query at hand and then, based on this information, will invoke the most appropriate technique for computing the consistent answers. Developing such a system, however, will require further exploration and deeper understanding of the boundary between tractability and intractability of consistent query answering.

Acknowledgment We would like to thank Monica Caniupan for her guidance in setting up ConsEx, and Ariel Fuxman for the help he provided with ConQuer.

8. REFERENCES

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP*, 3(4-5):393–424, 2003.

[3] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *TCS*, 296(3):405–434, 2003.

[4] P. Barceló and L. E. Bertossi. Logic programs for querying inconsistent databases. In *PADL*, pages 208–222, 2003.

[5] C. Beeri and R. Ramakrishnan. On the power of magic. In *JLP*, pages 269–283, 1987.

[6] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan and Claypool Publishers, 2011.

[7] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.

[8] M. Caniupán and L. E. Bertossi. The consistency extractor system: Querying inconsistent databases using answer set programs. In *SUM*, pages 74–88, 2007.

[9] M. Caniupán and L. E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *DKE*, 69(6):545–572, 2010.

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

[11] J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. In *CIKM*, pages 417–426, 2004.

[12] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A system for computing consistent answers to a class of sql queries. In *EDBT*, pages 841–844, 2004.

[13] T. Eiter, W. Faber, C. Koch, N. Leone, and G. Pfeifer. DLV - a system for declarative problem solving. *CoRR*, cs.AI/0003036, 2000.

[14] T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data integration systems. In *ICLP*, pages 163–177, 2003.

[15] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[16] S. Flesca, F. Furfaro, and F. Parisi. Consistent answers to Boolean aggregate queries under aggregate constraints. In *DEXA (2)*, pages 285–299, 2010.

[17] S. Flesca, F. Furfaro, and F. Parisi. Querying and repairing inconsistent numerical databases. *ACM TODS*, 35(2), 2010.

[18] S. Flesca, F. Furfaro, and F. Parisi. Range-consistent answers of aggregate queries under aggregate constraints. In *SUM*, pages 163–176, 2010.

[19] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD*, pages 155–166, 2005.

[20] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. *JCSS*, 73(4):610–635, 2007.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[22] G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *ICLP*, pages 348–364, 2001.

[23] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE TKDE*, 15(6):1389–1408, 2003.

[24] P. G. Kolaitis and E. Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85, 2012.

[25] N. Leone, T. Eiter, W. Faber, M. Fink, G. Gottlob, and G. Greco. Boosting information integration: The INFOMIX system. In *SEBD*, pages 55–66, 2005.

[26] D. V. Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. *TPLP*, 6(1-2):107–167, 2006.

[27] J. Wijsen. On the consistent rewriting of conjunctive queries under primary key constraints. *Inf. Syst.*, 34(7):578–601, 2009.

[28] J. Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *IPL*, 110(21):950–955, 2010.

[29] J. Wijsen. Certain conjunctive query answering in first-order logic. *ACM TODS*, 37(2):9, 2012.