On Condensing Database Repairs Obtained by Tuple Deletions

Jef Wijsen
Université de Mons-Hainaut
B-7000 Mons, Belgium
wijsen@umh.ac.be

Abstract

An inconsistent database is repaired by restricting it to a maximal consistent subset of its tuples. Typically, this gives rise to a multitude of possible repairs, and hence to a multitude of possible answers to a given query. The consistent (or certain) query answer consists of the tuples that are in all possible answers. For conjunctive queries, this consistent answer can also be obtained by asking the query once on a database that is maximal homomorphic to all repairs. We study the complexity of constructing this maximal homomorphic database for different classes of constraints and queries.

Keywords: Consistent query answer, database repair.

1. Introduction

Many operational databases contain data that are known or suspected to be inconsistent. Data integration is among the most cited sources of inconsistency: for example, two databases may each satisfy a primary key constraint, but their union may not. Consider the following relation $S$ of great sportsmen:

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill Russell</td>
<td>1934</td>
<td>Basketball</td>
</tr>
<tr>
<td>Bill Russell</td>
<td>1936</td>
<td>Basketball</td>
</tr>
<tr>
<td>J. Anquetil</td>
<td>1934</td>
<td>Cyclist</td>
</tr>
<tr>
<td>R. Poulidor</td>
<td>1936</td>
<td>Cyclist</td>
</tr>
</tbody>
</table>

Since no sportsman can have two distinct years of birth, the first two tuples contradict each other. These tuples may come from two different sources. The contradiction can be repaired by deleting either tuple, which yields two possible repairs $R_1$ and $R_2$:

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill Russell</td>
<td>1934</td>
<td>Basketball</td>
</tr>
<tr>
<td>J. Anquetil</td>
<td>1934</td>
<td>Cyclist</td>
</tr>
<tr>
<td>R. Poulidor</td>
<td>1936</td>
<td>Cyclist</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill Russell</td>
<td>1936</td>
<td>Basketball</td>
</tr>
<tr>
<td>J. Anquetil</td>
<td>1934</td>
<td>Cyclist</td>
</tr>
<tr>
<td>R. Poulidor</td>
<td>1936</td>
<td>Cyclist</td>
</tr>
</tbody>
</table>

In general, the number of repairs can be exponential in the size of the original database. Then, since every repair is equally possible, there is more than one possible answer to a given query. It is common practice to return only the certain query answer, i.e. the tuples that are in all possible query answers. For example, for the query “Give sportsmen born in 1934,” the repair $R_1$ returns both Bill Russell and J. Anquetil, while $R_2$ only returns J. Anquetil. Thus, only J. Anquetil is a certain answer to this query.

For conjunctive queries, all repairs can be replaced by a single database that, when queried, will return consistent answers. Such database obtained from “integrating” all repairs will be called nucleus. For the current example, the nucleus looks as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill Russell</td>
<td>$y$</td>
<td>Basketball</td>
</tr>
<tr>
<td>$x$</td>
<td>$y$</td>
<td>Cyclist</td>
</tr>
<tr>
<td>J. Anquetil</td>
<td>1934</td>
<td>Cyclist</td>
</tr>
<tr>
<td>R. Poulidor</td>
<td>1936</td>
<td>Cyclist</td>
</tr>
</tbody>
</table>

Note that in this representation, J. Anquetil is the only sportsman to be certainly born in the year 1934. Nuclei can (and mostly will) contain variables. The double occurrence of the variable $y$ models that in both repairs $R_1$ and $R_2$, Bill Russell is born in the same year as some cyclist (but we do not know which one). In this way, if we ask $S'$ to “Give basketball players that were born in the same year as some cyclist,” we get Bill Russell, as it should be, as Bill Russell is in the answer to this query on both repairs $R_1$ and $R_2$.

Although we know from previous work [18] how to effectively construct a nucleus, up to now, we had no clue about the tractability of this construction. The central questions in this paper are: “How big is the nucleus?” and “How
much time does it take to construct it?”. We first show that in general the size of the nucleus is not polynomially bounded in the size of the database to be repaired. We then identify restricted cases where the nucleus is polynomially bounded. Since the computation of a nucleus of polynomial size may still take exponential time, we finally identify cases where the nucleus construction is in polynomial time.

The paper is organized as follows. Section 2 starts by recalling some database constructs and then formalizes the concept of nucleus. Section 3 investigates the size of nuclei. Section 4 distinguishes conditions that allow a nucleus construction in polynomial time. The discussion in Section 5 relates our work to other approaches for consistent query answering.

2. Theoretical Setting

2.1. Database Homomorphism

For every \( n \in \mathbb{N} \), we assume denumerably many relation symbols of arity \( n \). An atom is, for example, \( S^0(x, y, \text{Cyclist}) \), and a database is a finite set of atoms.

**Definition 1** A database \( D \) is a finite set of (not necessarily ground) atoms. An atom or database without variables is called ground. A ground atom is also called a fact. For a database \( D \), we define \( \text{grd}(D) := \{ L \in D \mid L \text{ is ground} \} \).

The following definition introduces the lattice induced by the notion of database homomorphism.

**Definition 2** A homomorphism from database \( C \) to database \( D \) is a substitution \( \theta \) for the variables in \( C \) such that \( \theta(C) \subseteq D \). If such a homomorphism from \( C \) to \( D \) exists, then \( C \) is said to be homomorphic to \( D \), denoted \( C \models D \). Two databases are equivalent, denoted \( C \sim D \), iff \( C \models D \) and \( D \models C \).

**Let \( \mathcal{D} \) be a set of databases. An infimum of \( \mathcal{D} \) is a database \( C \) satisfying:**

1. \( C \) is homomorphic to each database in \( \mathcal{D} \); and
2. Maximality: Every database \( C' \) that is homomorphic to each database in \( \mathcal{D} \), is also homomorphic to \( C \).

For the example of Section 1, \( S \models R_1 \models S' \) and \( S \models R_2 \models S' \). To see why, note that the substitution \( \theta_1 = \{(x, \text{J. Anquetil}), (y, 1934)\} \) maps \( S' \) into \( R_1 \), and \( \theta_2 = \{(x, \text{R. Poulidor}), (y, 1936)\} \) maps \( S' \) into \( R_2 \). Given a finite set \( \mathcal{D} \) of databases, there is an effective way to construct an infimum of \( \mathcal{D} \). For the current paper, it is not important to know the construction itself. In the running example, \( S' \) is an infimum of \( \{R_1, R_2\} \). It is easy to see that the set of all infimums of \( \mathcal{D} \) forms a \( \sim \)-equivalence class. We assume there is a selection rule that picks an arbitrary representative from this equivalence class and denote it \( \inf(\mathcal{D}) \).

2.2. Commuting Diagram

A conjunctive query \( q \) is written in the form of a rule [1]:

\[
\text{ans}(\vec{x}) \leftarrow R_1(\vec{x}_1), R_2(\vec{x}_2), \ldots, R_k(\vec{x}_k),
\]

where each variable in \( \vec{x} \) occurs in some \( \vec{x}_i \). The term \( \text{rule} \) will be used as a shorthand for “rule-based conjunctive query.” The answer to a rule \( q \) on a (not necessarily ground) database \( D \) is denoted \( q(D) \). The class of rule-based conjunctive queries will be denoted \( \mathcal{CQ} \).

In earlier work [18], we showed that we have the following commutative diagram. Here, \( q \) is a conjunctive query, \( \mathcal{D} \) is a finite set of databases, and \( q(\mathcal{D}) \) is defined by \( q(\mathcal{D}) := \{ q(D) \mid D \in \mathcal{D} \} \).

\[
\begin{array}{ccc}
\mathcal{D} & \xrightarrow{\inf} & \inf(\mathcal{D}) \\
q \downarrow & & \downarrow \text{id}
\end{array}
\]

That is, the infimum of query answers on multiple databases is equivalent to the query answer on the infimum of the databases.

2.3. Consistent Query Answering

An inconsistent database is repaired by deleting a minimal (w.r.t. \( \subseteq \)) subset of its facts. We will require that integrity constraints be satisfied by the empty database, so as to guarantee that every database can be made consistent by deletions. So we exclude insertions as a repair primitive. Note that for certain constraints, like key dependencies, insertions are of no help in restoring consistency.

**Definition 3** Let \( \Sigma \) be a set of integrity constraints that is satisfied by the empty database. A repair of a ground database \( D \) and \( \Sigma \) is a database \( R \) satisfying:

1. \( R \not\subseteq D \) and \( R \models \Sigma \); and
2. Maximality: for every database \( R' \), if \( R \not\subseteq R' \subseteq D \), then \( R' \not\models \Sigma \).

The number of repairs of a database \( D \) is finite, but can be exponential in the size of \( D \). How do we answer to a query if there is more than one repair? Every repair gives a possible answer to the query. We are interested in the facts that are certainly in the answer, i.e. that are in all possible answers.

**Definition 4** Let \( D \) be a ground database and \( \Sigma \) a set of constraints that is satisfied by the empty database. Let \( q \) be a query. Let \( \mathcal{D} \) be the set of all repairs of \( D \) and \( \Sigma \). The consistent answer to the query \( q \) on input \( D \) and \( \Sigma \), denoted \( q_\Sigma(D) \), is defined by \( q_\Sigma(D) := \bigcap q(\mathcal{D}) \).\(^1\)

\(^1\)Recall that \( q(\mathcal{D}) := \{ q(D) \mid D \in \mathcal{D} \} \).
Let \( \Sigma \) be a set of constraints satisfied by the empty database, and \( q \) a conjunctive query. We define consistent query answering as the complexity of the set:

\[
CQA(\Sigma, q) := \{ D \mid D \text{ is a ground database and } q_\Sigma(D) = \{ \} \},
\]

i.e. the data complexity of testing whether \( q_\Sigma(D) \) is empty for a database \( D \). Clearly, computing the consistent answer itself is at least as hard as just testing its emptiness.

2.4. Nucleus

It is not hard to see that if \( \mathcal{D} \) is a finite set of ground databases, then the facts that are in every database of \( \mathcal{D} \) must necessarily be in \( \inf(\mathcal{D}) \). Conversely, every fact in \( \inf(\mathcal{D}) \) must appear in every database of \( \mathcal{D} \). So we have shown that \( \bigcap \mathcal{D} = \grd(\inf(\mathcal{D})) \). If we substitute \( q(\mathcal{D}) \) for \( \mathcal{D} \) in this equation, we obtain \( \bigcap q(\mathcal{D}) = \grd(\inf(q(\mathcal{D}))) \).

Then, using the commutative diagram of Section 2.2,

\[
\bigcap q(\mathcal{D}) = \grd(q(\inf(\mathcal{D}))).
\]

If we consider \( \mathcal{D} \) to be a set of repairs and read \( \bigcap q(\mathcal{D}) \) as consistent answer (see Def. 4), what this says is that the consistent answer to a conjunctive query can be obtained in three successive steps:

1. first, compute the infimum \( \inf(\mathcal{D}) \) of all repairs (call this infimum \( C \));
2. next, execute the query \( q \) on \( C \) giving \( q(C) \);
3. finally, return the ground tuples in the query answer, i.e. return \( \grd(q(C)) \).

If \( q \) is a conjunctive query, the second step takes only polynomial in the size of \( C \). The third step takes only linear time in the size of the query answer. So any non-polynomial complexity must come from the first step.

Significantly, the infimum computed in the first step does not depend on the query \( q \). This means that to answer multiple conjunctive queries, the first step needs to be executed only once. This motivates the following definition.

Definition 5 Let \( \mathcal{L} \) be a query language. Let \( D \) be a ground database and \( \Sigma \) a set of integrity constraints satisfied by the empty database. A (not necessarily ground) database \( C \) is called a \( \mathcal{L} \)-nucleus of \( D \) and \( \Sigma \) iff for every query \( q \) in \( \mathcal{L} \), \( q_\Sigma(D) = \grd(q(C)) \). We will use nucleus as a shorthand for \( \mathcal{CQ} \)-nucleus (i.e. \( \mathcal{CQ} \) is the language by default).

From what precedes, it is evident that every set of integrity constraints that is satisfied by the empty database allows a nucleus; moreover, this nucleus is unique up to \( \sim \)-equivalence.

**Figure 1. Database \( \text{db}^n \) where \( n \geq 2 \).**

**Theorem 1** Let \( \Sigma \) be a set of integrity constraints satisfied by the empty database. Let \( D \) be a ground database. Then,

1. There exists an effective procedure for constructing a \( \mathcal{CQ} \)-nucleus of \( D \) and \( \Sigma \).
2. If \( C_1, C_2 \) are \( \mathcal{CQ} \)-nuclei of \( D \) and \( \Sigma \), then \( C_1 \sim C_2 \).

Now that we know that nuclei exist, we turn to their construction complexity. Three scenarios can occur:

S1. The nucleus size is not polynomially bounded in the size of the database to be repaired.

S2. The nucleus size is polynomially bounded but constructing the nucleus is not in polynomial time.

S3. Constructing a nucleus is in polynomial time.

The scenario S3 is the most optimistic one: if it occurs, then consistent query answering is tractable, because querying the nucleus and removing nonground atoms from the answer takes only polynomial time. However, since Chomicki and Marcinkowski [10] showed that consistent query answering under tuple deletions quickly becomes intractable, we already know that this scenario is rather exceptional. In the next section, we show that even the worst case scenario S1 cannot be ruled out.

3. On the Size of Nuclei

We show that a nucleus can be exponentially large in general. We then look at restricted query classes that guarantee the existence of polynomial size nuclei.

3.1. Nucleus of Exponential Size

The following result is interesting but rather unpleasant.

**Theorem 2** Let \( n \geq 7 \). Let \( \text{db}^n \) be the database encoding the graph in Fig. 1; that is, \( \text{db}^n \) contains \( R(i, j) \) iff there is an edge from node \( i \) to node \( j \). Let \( \sigma = \forall x \forall y \forall z (R(x, y) \land R(x, z) \Rightarrow y = z) \). Then, every \( \mathcal{CQ} \)-nucleus of \( \text{db}^n \) and \( \{ \sigma \} \) contains at least \( 2^n \) atoms.

We next study two subclasses of rules that guarantee the existence of polynomial size nuclei.
3.2. Linear Rules

The class of linear rules is a strict subset of the class of rule-based conjunctive queries.

**Definition 6** A rule-based conjunctive query is linear if every variable that occurs in two distinct atoms of the rule body, also occurs in the head. The class of linear rules is denoted \( \text{linCQ} \).

We have the following result.

**Theorem 3** Let \( \Sigma \) a set of constraints that is satisfied by the empty database. For every ground database \( D \), there exists a \( \text{linCQ} \)-nucleus of \( D \) and \( \Sigma \) whose size is linearly bounded in \( |D| \).

3.3. Quantifier-free Rules

The class of quantifier-free rules is a strict subset of the class of linear rules.

**Definition 7** A rule-based conjunctive query is quantifier-free if every variable that occurs in the body of the rule, also occurs in the head. The set of quantifier-free rules is denoted \( \text{qfCQ} \).

Consequently, \( \text{qfCQ} \subseteq \text{linCQ} \subseteq \text{CQ} \). Theorem 4 parallels Theorem 3.

**Theorem 4** Let \( \Sigma \) a set of constraints that is satisfied by the empty database. For every ground database \( D \), there exists a subset \( C \subseteq D \) such that \( C \) is a \( \text{qfCQ} \)-nucleus of \( D \) and \( \Sigma \).

4. Nuclei Constructible in Polynomial Time

In the preceding section, we identified query classes that guarantee the existence of polynomial size nuclei. However, constructing a nucleus of polynomial size may still take exponential time. We now distinguish two cases of practical interest that allow nuclei computable in polynomial time: primary keys and denials.

Chomicki and Marcinkowski [10] showed intractability of consistent query answering for key dependencies and conjunctive queries. For linear rules, we know from Theorem 3 that a nucleus of polynomial size exists. Moreover, it is not hard to show that this \( \text{linCQ} \)-nucleus can be constructed in polynomial time, as stated by the next theorem.

**Theorem 5** Let \( \Sigma \) a set of key dependencies, at most one key dependency per relation symbol. For every ground database \( D \), constructing a \( \text{linCQ} \)-nucleus of \( D \) and \( \Sigma \) is in time \( O(n \log n) \) where \( n = |D| \).

Denials are Horn clauses with built-in predicates. From [10, Theorem 3.5], consistent query answering is intractable for denials, even for linear rules. This means that although polynomial size nuclei exist by Theorem 3, they cannot be constructed in polynomial time (unless \( P=NP \)). Tractability is obtained by requiring that rules be quantifier-free, as stated by the following theorem.

**Theorem 6** Let \( \Sigma \) a set of denials. Constructing a \( \text{qfCQ} \)-nucleus of \( D \) and \( \Sigma \) is in polynomial time in the cardinality of the input database \( D \).

5. Discussion

**Defining Repairs** Consistent query answering gathered much attention since the seminal paper by Arenas et al. [2]. An overview of database repairing and consistent query answering up to 2002 is [6].

Repairs are mostly defined in terms of the sets of inserted and deleted tuples. Insertions and deletions are mostly treated symmetrically [2]. An asymmetric treatment of insertions and deletions is the loosely-sound semantics introduced in [8], which minimizes (w.r.t. \( \subseteq \) ) the set of deleted tuples, irrespective of the tuples to be inserted. In [14], the symmetric treatment of insertions and deletions can be overridden by user-defined prioritized update rules. In the current paper, as in [10], only tuple deletions are allowed. Note incidentally that for key dependencies, tuple insertions are useless for restoring consistency.

Most approaches minimize sets of deleted and/or inserted tuples relative to set inclusion. Some authors also consider minimization w.r.t. cardinality [4, 5, 17]. Greco et al. [15] propose a formalism where preferences among repairs can be specified by a user-defined evaluation function.

**Computing Repairs and Consistent Query Answers** Several authors have investigated the use of logic solvers for database repairing [3, 5, 14]. They reduce database repairing and consistent query answering to the computation of models in some logical framework for which resolution methods exist. These papers focus more on expressiveness than on tractability.

Another elegant approach to compute consistent query answers is query rewriting. As a given query is rewritten such that the new query is guaranteed to return the consistent answer on any, possibly inconsistent, database. Obviously, if the rewritten query is wished to be first-order expressible [2, 9], then unless \( P=NP \), this approach is limited to sets \( \Sigma \) of constraints and queries \( q \) for which consistent query answering is in \( P \).

For example, assume a SQL table containing columns \( K \) and \( A \), where \( K \) is the primary key. Then, the SQL query:

\[
\text{SELECT DISTINCT A FROM R}
\]
may be rewritten as [11]:

```
SELECT DISTINCT R1.A FROM R AS R1
WHERE NOT EXISTS (
    SELECT * FROM R AS R2
```

It is easy to verify that the rewritten query gives us consistent query answers on possibly inconsistent databases. The original query can be expressed as a linear rule, and the time required to answer the rewritten query is no less than the time needed for constructing a \( \text{linCQ} \)-nucleus (i.e. \( O(n \log n) \) where \( n = |R| \), see Theorem 5). So in this example, nuclei are a valuable alternative to query rewriting. Moreover, the nucleus may be stored to answer subsequent \( \text{linCQ} \) queries as long as the underlying database is not modified.

Fuxman and Miller [13] give an algorithm for first-order rewriting of a subclass of conjunctive queries when there is at most one key dependency per relation, which implies tractability of consistent query answering under these conditions. The subclass they consider and our subclass \( \text{linCQ} \) are not comparable by set inclusion. So the tractability of consistent query answering implied by Theorem 5 is not implied by this earlier work.

In [11, 12], a practical implementation is presented for consistent query answering relative to denial constraints.

All approaches cited so far (including ours) assume a single database context. At the center of [7, 16] is the computation of consistent answers to queries on possibly inconsistent global databases that result from integrating local source databases.

References


