Jan Chomicki
Jef Wijsen (Eds.)

Inconsistency and Incompleteness in Databases

International Workshop Collocated with the 10th International Conference on Extending Database Technology

March 26, 2006, Munich (Germany)

Pre-proceedings
Inconsistency and Incompleteness in Databases

Papers from the EDBT 2006 Workshop
Preface

IIDB 2006, the International Workshop on Inconsistency and Incompleteness in Databases, was held on March 26, 2006, in Munich, Germany, as a collocated event of EDBT 2006, the 10th International Conference on Extending Database Technology. The workshop consisted of a keynote talk by Maurizio Lenzerini, followed by presentations of regular and position papers. This volume contains all regular and position papers presented at the workshop.

In response to the call for papers, ten papers were submitted. Each submission was reviewed by at least three program committee members. Five papers were accepted for regular and three for short, position paper presentation. Additionally, program committee members were invited to contribute position papers describing their ongoing research. Five such contributions are included in this volume.

The regular papers will be published by Springer-Verlag in the series Lecture Notes in Computer Science (LNCS), in a volume containing selected articles of all EDBT 2006 workshops.

February 2006

Jan Chomicki
Jef Wijen
Program Committee

Chairpersons

Jan Chomicki, University at Buffalo, USA
Jef Wijsen, University of Mons-Hainaut, BEL

Program Committee

Marcelo Arenas, PUC Chile, CHL
Ofer Arieli, Academic College of Tel-Aviv, ISR
Leopoldo Bertossi, Carleton University, CAN
Patrick Bosc, IRISA/ENSSAT, FRA
Andrea Cali, Free University of Bozen-Bolzano, ITA
Nilesh Dalvi, University of Washington, USA
Thomas Eiter, Vienna University of Technology, AUT
Wenfei Fan, University of Edinburgh, UK & Bell Labs, USA
Enrico Franconi, Free University of Bozen-Bolzano, ITA
Ariel Fuxman, University of Toronto, CAN
Gösta Grahne, Concordia University, CAN
Sergio Greco, University of Calabria, ITA
Maurizio Lenzerini, University of Rome “La Sapienza”, ITA
Jerzy Marcinkowski, Wroclaw University, POL
V.S. Subrahmanian, University of Maryland, USA
Table of Contents

Keynote Talk
Inconsistency Tolerance in P2P Data Integration
  M. Lenzerini ................................................................. 1

Regular Papers
DART: a Data Acquisition and Repairing Tool
  B. Fazzinga, S. Flesca, F. Furfaro, F. Parisi  ....................... 2
On the First-order Reducibility of Unions of Conjunctive Queries over Inconsistent Databases
  D. Lembo, R. Rosati, M. Ruzzi ........................................ 17
Semantically Correct Query Answers in the Presence of Null Values
  L. Bravo, L. Bertossi .................................................. 33
Models for Incomplete and Probabilistic Information
  T.J. Green, V. Tannen .................................................. 48
Preference-Driven Querying of Inconsistent Relational Databases
  S. Staworka, J. Chomicki, J. Marcinkowski  ......................... 67

Position Papers
Taming Data Explosion in Probabilistic Information Integration
  A. de Keijzer, M. van Keulen, Y. Li .................................. 82
Model Theoretic and Fixpoint Semantics for Preference Queries over Imperfect Data
  P. Vojtas ................................................................. 87
Consistency Management Framework for Realtime Disaster Information Systems
  I. Noda ........................................................................ 92
Dealing with Inconsistencies and Incompleteness in Database Update
  G. De Giacomo, M. Lenzerini, A. Poggi, R. Rosati ................ 97
Querying Incomplete Data: Towards Practical Cases
  A. Culi ................................................................. 99
A Note on Database Repairing by Value Modification
  J. Wijser ................................................................. 104
Some Research Directions in Consistent Query Answering: a Vision
  L. Bertossi ................................................................. 109
About Queries Addressed to Possibilistic Databases
  P. Bose, O. Pivert ....................................................... 114
Inconsistency Tolerance in P2P Data Integration

Maurizio Lenzerini

DASI-lab: Data and Service Integration Laboratory
Dipartimento di Informatica e Sistemistica “Antonio Ruberti”
Università degli Studi di Roma “La Sapienza”

Abstract. Integrating heterogeneous data which are distributed over the network is one of the crucial challenges at the current evolutionary stage of information technology infrastructures. Most of the formal approaches to data integration refer to an architecture based on a global schema and a set of sources. As observed in several contexts, this centralized architecture is not the best choice for supporting data integration, cooperation and coordination in highly dynamic computer networks. A more appealing architecture is the one based on peer-to-peer systems. In these systems every peer acts as both client and server, and provides part of the overall information available from a distributed environment, without relying on a single global view. Since peers represent interconnected autonomous information systems, one of the major challenges in the design of a well-founded peer-to-peer data integration system is to cope with mutual inconsistencies between data stored in the peers. In this talk we first review the principles of peer-to-peer data integration, and then we address the issue of inconsistency tolerance, by discussing the problem of assigning semantics to the data integration system in the presence of inconsistencies, and by illustrating possible approaches to answering queries coherently with such semantics.
DART: a Data Acquisition and Repairing Tool

Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Francesco Parisi

DEIS - Università della Calabria
Via Bucci - 87036 Rende (CS) ITALY
{bfazzinga, flesca, furfaro, fparisi}@deis.unical.it

Abstract. An architecture is proposed providing robust data acquisition facilities from input documents containing tabular data. This architecture is based on a data-repairing framework exploiting integrity constraints defined on the input data to support the detection and the repair of inconsistencies in the data arising from errors occurring in the acquisition phase. In particular, a specific but expressive form of integrity constraints (steady aggregate constraints) is defined which enables the computation of a repair to be expressed as a mixed integer linear programming problem.

1 Introduction

The need to acquire data from different sources of information often arises in many application scenarios, such as e-procurement, competitor analysis, business intelligence. In several cases these sources are heterogeneous documents, possibly represented according to different formats, ranging from paper documents to electronic ones (PDF, MSWord, HTML files). In order to be exploited to provide valuable knowledge, information must be extracted from the original documents and re-organized into a machine-readable format. The problem of defining efficient and effective approaches accomplishing this task is a challenging issue in the context of Information Extraction (IE).

Most of traditional IE techniques focus on efficiency, providing unsupervised extraction algorithms which automatically extract records from documents. However, it frequently happens that some of the extracted records are not correctly recognized, i.e. the value of one (or more) field has been misspelled. In several contexts (such as balance analysis) extracted information must be 100% error free in order to be profitably exploited, thus unsupervised approaches are not well-suited. In these cases, data transcription from input documents into a machine-readable format requires massive human intervention, thus compromising efficiency and making valuable resources be wasted. Human intervention is mainly devoted to verifying the correctness of acquired data by comparing them with the content of source documents.

Indeed, if integrity constraints are defined on the input data, this kind of human intervention can be reduced by automatically verifying whether acquired data satisfy these constraints, thus limiting manual corrections to those pieces of acquired data which do not satisfy them. In fact current approaches exploiting integrity constraints on source documents require inconsistent acquired data to be manually edited by a human operator. This editing task is likely to be onerous, since a large amount of data in the input documents need to be accessed and compared with the acquired ones.
The idea underlying this paper is that human intervention can be reduced by exploiting some repairing technique to suggest the “most likely” way of fixing inconsistent data. We introduce the architecture of a system (namely, DART - Data Acquisition and Repairing Tool) based on this idea. The motivation of this work and the contribution provided by this system can be better understood after reading the following example, describing a specific application scenario (that is, data acquisition from balance sheets).

*Example 1.* The balance sheet is a financial statement of a company providing information on what the company owns (its assets), what it owes (its liabilities), and the value of the business to its stockholders. A thorough analysis of a company balance sheet is extremely important for both stock and bond investors, since it allows potential liquidity problems to be detected, thus determining the company financial reliability as well as its ability to satisfy financial obligations.

Figure 1 is a portion of a document containing two *cash budgets* for a firm, each of them related to a year. Each cash budget is a summary of cash flows (receipts, disbursements, and cash balances) over the specified periods.

<table>
<thead>
<tr>
<th>Year</th>
<th>Receipts</th>
<th>2003</th>
<th>Disbursements</th>
<th>Balance</th>
<th>2004</th>
<th>Disbursements</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beginning cash</td>
<td>20</td>
<td>payment of accounts</td>
<td>120</td>
<td>60</td>
<td>payment of accounts</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>cash sales</td>
<td>100</td>
<td>capital expenditure</td>
<td>0</td>
<td>60</td>
<td>capital expenditure</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>receivables</td>
<td>120</td>
<td>long-term financing</td>
<td>40</td>
<td>ending cash balance</td>
<td>80</td>
<td>ending cash balance</td>
</tr>
<tr>
<td></td>
<td>total cash receipts</td>
<td>220</td>
<td>total disbursements</td>
<td>160</td>
<td></td>
<td>total disbursements</td>
<td>190</td>
</tr>
</tbody>
</table>

*Fig. 1.* An input document

This cash budget satisfies the following integrity constraints:

a) for each year, the sum of *cash sales* and *receivables* in section *Receipts* must be equal to *total cash receipts*;

b) for each year, the sum of *payment of accounts*, *capital expenditure* and *long-term financing* must be equal to *total disbursements* (in section *Disbursements*);

c) for each year, the *net cash inflow* must be equal to the difference between *total cash receipts* and *total disbursements*;

d) for each year, the *ending cash balance* must be equal to the sum of the *beginning cash* and the *net cash inflow*;

Generally balance sheets like the ones depicted in Figure 1 are available as paper documents, thus they cannot be automatically processed by balance analysis tools, since these work only on electronic data. In fact, some companies do business acquiring electronic balance data and reselling them in a format suitable for being processed by commercial analysis tools. Currently electronic versions are obtained by means of either human transcriptions or OCR acquisition tools. Both these approaches are likely to result in erroneous acquisition, thus compromising the reliability of the analysis task.

An example of numerical value recognition error occurring during the acquisition phase is the recognition of the value 250 instead of 220 for “total cash receipts” in the year 2003. Consequently, constraints a) and c) are not satisfied on the acquired data.
for year 2003. Furthermore, some symbol recognition errors in non-numerical strings may occur in the acquisition phase. For instance, the item “bguning cesh” could be recognized instead of “beginning cash”.

DART is a system supporting the acquisition of heterogeneous documents and the supervised repairing of the acquired data. With respect to Example 1, DART will suggest to change the “total cash receipts” value for year 2003 from 250 (i.e. the acquired value) to 220, thus reducing the human intervention, as the human operator is no longer required to access the whole input document to fix acquisition errors making integrity constraints violated. In particular, DART is based on the notion of card-minimal repair introduced in [11], where the problem of repairing numerical data which are inconsistent w.r.t. aggregate constraints is addressed. Aggregate constraints defined in [11] can express constraints like those defined in the context of balance-sheet data. The notion of card-minimal repair is well-suited for our context, where data inconsistency is due to bad symbol recognition during the acquisition phase. Indeed, applying the card-minimal semantics means searching for repairs changing the minimum number of acquired values, which corresponds to the assumption that the minimum number of errors occurred in the acquisition phase.

This work stems from a specific application context, where data to be acquired are balance sheets. In this scenario, the relevant information is formatted according to a tabular layout. Therefore, our acquisition approach is targeted to tabular data. However, observe that this feature does not limit DART to the acquisition of balance sheets, as tabular data often occur in many different application contexts, such as web sites publishing product catalogs.

Related Work

The most widely used notion of repair and consistent query answer on inconsistent data is that of [2]. Different approaches to the problem of extracting reliable information from inconsistent data had been introduced in [1, 7]. Based on the notions of repair and consistent query answer introduced in [2], several works [3, 4, 12] investigated more expressive classes of queries and constraints. In [8, 9] a practical framework for computing consistent query answer for relational database has been presented. All the above-cited approaches assume that tuple insertions and deletions are the basic primitives for repairing inconsistent data. More recently, in [10] a repairing strategy using only tuple deletions was proposed, and in [6, 16, 17] repairs consisting of also value-update operations were considered. In [5] the problem of repairing databases by fixing numerical data at attribute level was investigated in presence of both denial constraints and a non-linear form of multi-attribute aggregate constraints. In [11] the problem of repairing and extracting reliable information from data violating a given set of aggregate constraints was investigated. These constraints consist of linear inequalities on aggregate-sum queries issued on measure values stored in the database. This syntactic form enables meaningful constraints to be expressed, such as those of Example 1 as well as other forms which often occur in practice. In this work we define a restricted class of aggregate constraints and provide a method to compute a card-minimal repair defined in [11] (according to the card-minimal semantics, a repaired database $D'$ minimally differs from the original database $D$ iff the number of value updates yielding $D'$...
is minimum w.r.t. all other possible repairs). We exploit this computation method in the DART system where data are acquired by means of an acquisition tool and information is extracted and transformed by a wrapping system. Due to space limitations, we do not discuss existing wrapping techniques here; a detailed survey can be found in [14]. We only point out that the wrapping technique embedded into our system differs from the state of the art as our approach is mainly focused on data represented into tables with complex structure.

**Main contributions**

In this work we introduce a system architecture aiming at supervised acquiring of information encoded into tabular data inside documents with possibly heterogeneous formats. Main novelties of our proposal are the following:

1. Our system embeds a wrapping module for extracting information from tabular data. This module can manage tables having “variable” structures, i.e. tables whose cells can span multiple rows and columns, according to no pre-determined scheme. This is a valuable feature, as all existing wrapping techniques do not work at all or are far from being satisfactory on tabular data without a “rigid” structure.

2. A framework for computing card-minimal repairs on wrongly acquired data is introduced to drive the data validation process. This framework exploits a specific form of aggregate constraints (namely, *steady aggregate constraints*) defined on the source documents to check the consistency of the acquired data and computing a repair.

Describing our wrapping technique in detail is out of the scope of this paper. Here we will focus on presenting the architecture of our system and the technique adopted for computing repairs.

## 2 DART in a nutshell

DART (*Data Acquisition and Repairing Tool*) is a system providing robust data acquisition facilities. It takes as input documents containing tabular data, and it exploits integrity constraints defined on the input data to support the detecting and the repairing of inconsistencies due to errors occurring in the acquisition phase. If acquisition errors are detected, the system proposes a way to correct these errors. Proposed corrections are validated by means of human intervention. In order to detect and repair inconsistencies, integrity constraints are considered expressing algebraic relations among the numerical data reported in the cells of the input tables. These constraints are exploited only to fix the acquired numerical values. Moreover, a dictionary of the terms used in the specific scenario which the input documents refer to is exploited to provide spelling error corrections on non-numerical strings.

Two kinds of user interact with DART, namely the *acquisition designer* and the *operator*. The former is an expert on the application context and specifies the metadata which are used to support both the extraction of tabular data and the repairing process. The latter interacts with the system during the acquisition of each document: if the acquired data need to be corrected, he is prompted to validate proposed corrections.

As shown in Figure 2, DART consists of two macro-modules. The first module takes as input documents containing tabular data and returns a relational database where the
extracted tabular data are stored. It performs three steps: it loads the input document and convert it in HTML format, it extracts the tabular data from the HTML document and it transforms them into a database instance. This module exploits metadata specified by the acquisition designer, which describe the structure and the semantics of the input documents.

The second module takes as input the database instance $D$ generated by the acquisition and extraction module. It locates possible inconsistencies in $D$ and returns a repair for $D$. Both the inconsistency detection and the repair computation are accomplished according to a set of aggregate constraints $AC$ defined by acquisition designer and represented in the metadata. In more detail, the repairing module transforms the problem of finding a card-minimal repair for $D$ w.r.t. $AC$ into an MILP instance (Mixed-Integer Linear Programming problem) and solves it providing a repair for $D$. The proposed repair is then validated by the operator, who either accepts it or requires to compute a different repair. In fact, it can be the case that the proposed repair is unsatisfactory since the operator realizes that it consists of value updates which do not correspond to the actual content of the source document. In this case the operator inserts further constraints on the acquired data. Basically, he drives the repairing process by specifying the exact values that some pieces of the repaired data must take.

3 Preliminaries

We assume classical notions of database scheme, relational scheme, and relations. In the following we will also use a logical formalism to represent relational databases, and relational schemes will be represented by means of sorted predicates of the form $R(A_1: \Delta_1, \ldots, A_n: \Delta_n)$, where $A_1, \ldots, A_n$ are attribute names and $\Delta_1, \ldots, \Delta_n$ are the corresponding domains. Each $\Delta_i$ can be either $\mathbb{Z}$ (infinite domain of integers), $\mathbb{R}$ (reals), or $\mathbb{S}$ (strings). Domains $\mathbb{R}$ and $\mathbb{Z}$ will be said to be numerical domains, and attributes defined over $\mathbb{R}$ or $\mathbb{Z}$ will be said to be numerical attributes. Given a ground atom $t$ denoting a tuple, the value of attribute $A$ of $t$ will be denoted as $t[A]$.

Given a database scheme $D$, we will denote as $\mathcal{M}_D$ (namely, Measure attributes) the set of numerical attributes representing measure data. That is, $\mathcal{M}_D$ specifies the set of attributes representing measure values, such as weights, lengths, prices, etc. For instance, in Figure 3, $\mathcal{M}_D$ consists of the only attribute $Value$.

3.1 Aggregate constraints

Given a relational scheme $R(A_1: \Delta_1, \ldots, A_n: \Delta_n)$, an attribute expression on $R$ is defined recursively as follows:
- a numerical constant is an attribute expression;
- each $A_i$ (with $i \in \{1..n\}$) is an attribute expression;
- $e_1 \psi e_2$ is an attribute expression on $R$, if $e_1$, $e_2$ are attribute expressions on $R$ and $\psi$ is an arithmetic operator in \{+,-\};
- $c \times (e)$ is an attribute expression on $R$, if $e$ is an attribute expression on $R$ and $c$ a numerical constant.

Let $R$ be a relational scheme and $e$ an attribute expression on $R$. An aggregation function on $R$ is a function $\chi : (A_1 \times \cdots \times A_k) \to \mathbb{R}$, where each $A_i$ is either $\mathbb{Z}$, or $\mathbb{R}$, or $\mathbb{S}$, and it is defined as follows:

$$\chi(x_1, \ldots , x_k) = \text{SELECT } \text{sum}(e) \text{ FROM } R \text{ WHERE } \alpha(x_1, \ldots , x_k)$$

where $\alpha(x_1, \ldots , x_k)$ is a boolean formula on $x_1, \ldots , x_k$, constants and attributes of $R$.

**Example 2.** Consider the database scheme $\mathcal{D}$ consisting of the single relation scheme $\text{CashBudget}($\text{Year}, \text{Section}, \text{Subsection}, \text{Type}, \text{Value})$, and its instance reported in Figure 3. This instance represents a possible output of the acquisition and extraction module when DART takes as input the document in Figure 1 (it results from the case that a symbol recognition error occurred in the acquisition phase, so that the acquired value of total cash receipts is 250 instead of 220). Values ‘det’, ‘aggr’ and ‘drv’ in column $\text{Type}$ stand for detail, aggregate and derived, respectively. In particular, an item of the table is aggregate if it is obtained by aggregating items of type detail of the same section, whereas a derived item is an item whose value can be computed using the values of other items of any type and belonging to any section.

<table>
<thead>
<tr>
<th>Year</th>
<th>Section</th>
<th>Subsection</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>Receipts</td>
<td>drv</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Receipts</td>
<td>det</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Receipts</td>
<td>det</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Receipts</td>
<td>agg</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Disbursements</td>
<td>det</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Disbursements</td>
<td>agg</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td>Balance</td>
<td>drv</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Receipts</td>
<td>agg</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Receipts</td>
<td>det</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Receipts</td>
<td>det</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Disbursements</td>
<td>det</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Disbursements</td>
<td>det</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Disbursements</td>
<td>agg</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Disbursements</td>
<td>agg</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Balance</td>
<td>drv</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>Balance</td>
<td>drv</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 3.** A cash budget

The following aggregation functions are defined on relational scheme $\text{CashBudget}$:

$$\chi_1(x, y, z) = \text{SELECT } \text{sum}(\text{Value}) \text{ FROM } \text{CashBudget} \text{ WHERE } \text{Section}=x \text{ AND } \text{Year}=y \text{ AND } \text{Type}=z$$

$$\chi_2(x, y) = \text{SELECT } \text{sum}(\text{Value}) \text{ FROM } \text{CashBudget} \text{ WHERE } \text{Year}=x \text{ AND } \text{Subsection}=y$$

Function $\chi_1$ returns the sum of $\text{Value}$ of all the tuples having $\text{Section}$ $x$, $\text{Year}$ $y$ and $\text{Type}$ $z$. For instance, $\chi_1(\text{‘Receipts’}, 2003, \text{‘det’})$ returns $100 + 120 = 220$, whereas $\chi_1(\text{‘Disbursements’}, 2003, \text{‘agg’})$ returns 160. Function $\chi_2$ returns the sum of $\text{Value}$ of all the tuples where $\text{Year}=x$ and $\text{Subsection}=y$. In our running example, as the pair
**Year, Subsection** is a key for the tuples of CashBudget, the sum returned by $\chi_2$ is an attribute value of a single tuple. For instance, $\chi_2(‘2003’, ‘cash sales’) returns 100, whereas $\chi_2(‘2004’, ‘net cash inflow’) returns 10.

**Definition 1 (Aggregate constraint).** Given a database scheme $D$, an aggregate constraint on $D$ is an expression of the form:

$$\forall x_1, \ldots, x_k \left( \phi(x_1, \ldots, x_k) \implies \sum_{i=1}^{n} c_i \cdot \chi_i(X_i) \leq K \right) \tag{1}$$

where:
1. $c_1, \ldots, c_n, K$ are constants;
2. $\phi(x_1, \ldots, x_k)$ is a conjunction of atoms containing the variables $x_1, \ldots, x_k$;
3. each $\chi_i(X_i)$ is an aggregation function, where $X_i$ is a list of variables and constants, and variables appearing in $X_i$ are a subset of $\{x_1, \ldots, x_k\}$.

Given a database $D$ and a set of aggregate constraints $AC$, we will use the notation $D \models AC$ [resp. $D \not\models AC$] to say that $D$ is consistent [resp. inconsistent] w.r.t. $AC$.

Observe that aggregate constraints enable equalities to be expressed as well, since an equality can be viewed as a pair of inequalities. For the sake of brevity, in the following equalities will be written explicitly.

**Example 3.** Constraints a) and b) defined in Example 1 can be expressed as: for each section and year, the sum of the values of all detail items must be equal to the value of the aggregate item of the same section and year, that is:

**Constraint 1:**
$$\forall x, y, s, t, v \text{ CashBudget}(y, x, s, t, v) \implies \chi_1(x, y, ‘det’) − \chi_1(x, y, ‘aggr’) = 0$$

For the sake of simplicity, in the following we will use a shorter notation for denoting aggregate constraints, where universal quantification is implied and variables in $\phi$ which do not occur in any aggregation function are replaced with the symbol ‘.’. For instance, Constraint 1 of Example 3 can be written as:

$$\text{CashBudget}(y, x, ., ., .) \implies \chi_1(x, y, ‘det’) - \chi_1(x, y, ‘aggr’) = 0$$

**Example 4.** Constraints c) and d) of Example 1 can be expressed as follows:

**Constraint 2:**
$$\chi_2(x, ‘net cash inflow’) - (\chi_2(x, ‘total cash receipts’) - \chi_2(x, ‘total disbursements’)) = 0$$

**Constraint 3:**
$$\chi_2(x, ‘ending cash balance’) - (\chi_2(x, ‘beginning cash’) + \chi_2(x, ‘net cash balance’)) = 0$$

### 3.2 Repairing inconsistent databases

Updates at attribute-level will be used in the following as the basic primitives for repairing data violating aggregate constraints. Given a relational scheme $R$ in the database scheme $D$, let $M_R = \{A_1, \ldots, A_k\}$ be the subset of $M_D$ containing all the attributes in $R$ belonging to $M_D$.

**Definition 2 (Atomic update).** Let $t = R(v_1, \ldots, v_n)$ be a tuple on the relational scheme $R(A_1 : \Delta_1, \ldots, A_n : \Delta_n)$. An atomic update on $t$ is a triplet $< t, A_i, v_i' >$, where $A_i \in M_R$ and $v_i'$ is a value in $\Delta_i$ and $v_i' \neq v_i$. 


Update $u = (<t, A_i, v_i'>)$ replaces $t[A_i]$ with $v_i'$, thus yielding the tuple $u(t) = R(v_1, \ldots, v_{i-1}, v_i', v_{i+1}, \ldots, v_n)$.

Observe that atomic updates work on the set $\mathcal{M}_R$ of measure attributes, as our framework is based on the assumption that data inconsistency is due to errors in the acquisition phase. Therefore we only consider repairs aiming at re-constructing the correct measure data.

Given an update $u$, we denote the pair $<\text{tuple, attribute}>$ updated by $u$ as $\lambda(u)$. That is, if $u = (<t, A_i, v>)$ then $\lambda(u) = (<t, A_i>)$.

**Definition 3 (Consistent database update).** Let $D$ be a database and $U = \{u_1, \ldots, u_n\}$ be a set of atomic updates on tuples of $D$. The set $U$ is said to be a consistent database update iff $\forall j, k \in [1..n] \text{ if } j \neq k \text{ then } \lambda(u_j) \neq \lambda(u_k)$.

Informally, a set of atomic updates $U$ is a consistent database update iff for each pair of updates $u_1, u_2 \in U$, $u_1$ and $u_2$ do not work on the same tuples, or they change different attributes of the same tuple.

The set of pairs $<\text{tuple, attribute}>$ updated by a consistent database update $U$ will be denoted as $\lambda(U) = \cup_{u \in U} \lambda(u_i)$.

Given a database $D$ and a consistent database update $U$, performing $U$ on $D$ results in the database $U(D)$ obtained by applying all atomic updates in $U$.

**Definition 4 (Repair).** Let $D$ be a database scheme, $\mathcal{AC}$ a set of aggregate constraints on $D$, and $D$ an instance of $D$ such that $D \not\models \mathcal{AC}$. A repair $\rho$ for $D$ is a consistent database update such that $\rho(D) \models \mathcal{AC}$.

**Example 5.** A repair $\rho$ for $\text{CashBudget}$ w.r.t. constraints 1), 2) and 3) consists in decreasing attribute $\text{Value}$ in tuple: $t = \text{CashBudget}(2003, \text{‘Receipts’, ‘total cash receipts’, ‘aggr’, 250})$ down to 220; that is, $\rho = \{ (<t, \text{Value}, 220>) \}$. $\square$

If a repair exists, different repairs can be performed on $D$ yielding a new database consistent w.r.t. $\mathcal{AC}$, although not all of them can be considered “reasonable”. For instance, if a repair exists for $D$ changing only one value in one tuple of $D$, any repair updating all values in all tuples of $D$ can be reasonably disregarded. To evaluate whether a repair should be considered “relevant” or not, we use an ordering criteria stating that a repair $\rho_1$ is preferred w.r.t. a repair $\rho_2$ if the number of changes issued by $\rho_1$ is less than $\rho_2$.

**Definition 5 (Card-minimal repair).** Let $D$ be a database scheme, $\mathcal{AC}$ a set of aggregate constraints on $D$, and $D$ an instance of $D$. A repair $\rho$ for $D$ w.r.t. $\mathcal{AC}$ is a card-minimal repair iff there is no repair $\rho'$ for $D$ w.r.t. $\mathcal{AC}$ such that $|\lambda(\rho')| < |\lambda(\rho)|$.

**Example 6.** Repair $\rho$ of Example 5 is a card-minimal repair. $\square$

Given a database $D$ which is not consistent w.r.t. a set of aggregate constraints $\mathcal{AC}$, different card-minimal repairs can exist on $D$. In our running example, repair $\rho$ of Example 5 is the unique card-minimal repair.

In [11] the problem of repairing and extracting reliable information from data violating a given set of aggregate constraints has been investigated. It has been shown that
1) given a database $D$ violating a set of aggregate constraints, deciding whether a repair for $D$ exists is NP-complete, and 2) given a database $D$ violating a set of aggregate constraints and a repair $\rho$ for $D$, deciding whether $\rho$ is a card-minimal repair is coNP-complete. Furthermore, the consistent query answer under both the set-minimal and the card-minimal semantic has been studied.

4 Steady aggregate constraints

In this section we introduce a restricted form of aggregate constraints, namely steady aggregate constraints. On the one hand, steady aggregate constraints are less expressive than (general) aggregate constraints, but, on the other hand, computing a card-minimal repair w.r.t. a set of steady aggregate constraints can be accomplished by solving an instance of an MILP (Mixed Integer Linear Programming) problem. This allows us to adopt standard techniques addressing MILP problems to accomplish the computation of a card-minimal repair (as it will be clear in the following, this would not be possible for general aggregate constraints). However, observe that the loss in expressiveness is not dramatic, as steady aggregate constraints suffice to express relevant integrity constraints in many real-life scenarios. For instance, all the aggregate constraints introduced in our running example can be expressed by means of steady aggregate constraints.

Before providing the formal definition of steady aggregate constraint, we introduce some preliminary notations.

Given a relational scheme $R(A_1, \ldots, A_n)$ and a conjunction of atoms $\phi$ containing the atom $R(x_1, \ldots, x_n)$, we say that the attribute $A_j$ corresponds to the variable $x_j$, for each $j \in [1..n]$. Given an aggregation function $\chi_i$, we will denote as $\mathcal{W}(\chi_i)$ the union of the set of the attributes appearing in the WHERE clause of $\chi_i$ and the set of attributes corresponding to variables appearing in the WHERE clause of $\chi_i$. Given an aggregate constraint $\kappa$ where the aggregation functions $\chi_1, \ldots, \chi_n$ occur, we will denote as $\mathcal{A}(\kappa)$ the set of attributes $\bigcup_{i=1}^{n} \mathcal{W}(\chi_i)$. Given an aggregate constraint $\kappa$, we will denote as $\mathcal{J}(\kappa)$ the set of attributes such that for each $A \in \mathcal{J}(\kappa)$ there are two atoms $R_i(x_{i1}, \ldots, x_{in})$ and $R_j(x_{j1}, \ldots, x_{jm})$ in $\phi(x_1, \ldots, x_k)$ satisfying both the following conditions:

1. there are $i_l \in [i_1..i_n]$ and $j_h \in [j_1..j_m]$ such that $x_{il} = x_{jh}$;
2. $A$ corresponds to either $x_{il}$ or $x_{jh}$.

Basicall, $\mathcal{J}(\kappa)$ contains attributes $A$ corresponding to variables shared by two atoms in $\phi$. The reason why sets $\mathcal{A}(\kappa)$ and $\mathcal{J}(\kappa)$ have been introduced is that they allow us to detect a useful property. In fact, in the case that $\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)$ does not contain any measure attribute, the tuples in the database instance $D$ which are “involved” in $\kappa$ (i.e. the tuples where $\phi$ and the WHERE clauses of the aggregation functions in $\kappa$ evaluate to true) can be detected without looking at the values of their measure attributes. As it will be clear in the following, if this syntactic property holds we can translate $\kappa$ into a set of linear inequalities and then express the computation of a card-minimal repair w.r.t. $\kappa$ as an instance of MILP.

**Definition 6 (Steady aggregate constraint).** Let $D$ be a database scheme, $\mathcal{M}_D$ the set of measure attributes of $D$ and $\kappa$ an aggregate constraint on $D$. An aggregate constraint $\kappa$ is said to be a steady aggregate constraint if:

$$(\mathcal{A}(\kappa) \cup \mathcal{J}(\kappa)) \cap \mathcal{M}_D = \emptyset \quad (2)$$
Example 7. Let \( D \) be a database scheme containing relational schemes \( R_1(A_1, A_2, A_3) \) and \( R_2(A_4, A_5, A_6) \), where \( \mathcal{M}_D = \{ A_2, A_4 \} \). Let \( \kappa \) be the following aggregate constraint on \( D \):
\[
\forall x_1, x_2, x_3, x_4, x_5 \ (R_1(x_1, x_2, x_3), R_2(x_3, x_4, x_5) \implies \chi(x_2) \leq K)
\]
where: \( \chi(x) = \text{SELECT sum}(A_6) \)
\[
\text{FROM } R_2
\]
\[
\text{WHERE } A_5 = x
\]
We have that \( \mathcal{A}(\kappa) = \{ A_5, A_2 \} \) and \( \mathcal{J}(\kappa) = \{ A_3, A_4 \} \), therefore \( \kappa \) is not a steady aggregate constraint.

Consider Constraint 1 of our running example. We have that \( \mathcal{A}(\text{Constraint 1}) = \{ \text{Year, Section, Type} \} \) and \( \mathcal{J}(\text{Constraint 1}) = \emptyset \). Since \( \mathcal{M}_D = \{ \text{Value} \} \), Constraint 1 is a steady aggregate constraint. Similarly, it is straightforward to show that also constraints 2) and 3) are steady aggregate constraints.

5 Computing a card-minimal repair

It can be shown that all complexity results (characterizing either the repair existence problem and the consistent query answer problem) given in [11] (where general aggregate constraints were considered) are still valid for our restricted class of aggregate constraints. Here we provide a technique for computing a card-minimal repair in presence of steady aggregate constraints exploiting the restrictions imposed on this form of constraints. As it will be clear later, this approach does not work for (general) aggregate constraints.

Consider a database scheme \( D \) and a set of steady aggregate constraints \( \mathcal{AC} \) on \( D \). In this case, we can model the problem of finding a card-minimal repair as a mixed-integer linear programming problem (MILP) (if the domain of numerical attributes is restricted to \( \mathbb{Z} \) then it can be formulated as an ILP problem).

We first show how a steady aggregate constraint can be expressed by a set of linear inequalities. Consider the steady aggregate constraint \( \kappa \):
\[
\forall x_1, \ldots, x_k \ \left( \phi(x_1, \ldots, x_k) \implies \sum_{i=1}^{n} c_i \cdot \chi_i(y_{i_1}, \ldots, y_{i_m}) \leq K \right)
\]
where \( \bigcup_{i=1}^{n} \{ y_{i_1}, \ldots, y_{i_m} \} \) is a subset of \( \{ x_1, \ldots, x_k \} \) and for each \( i \in [1..n] \):
\[
\chi_i(y_{i_1}, \ldots, y_{i_m}) = \text{SELECT sum}(e_i) \)
\[
\text{FROM } R_{A_i}
\]
\[
\text{WHERE } \theta_{A_i}(y_{i_1}, \ldots, y_{i_m})
\]
Without loss of generality, we assume that each attribute expression \( e_i \) occurring in the aggregation function \( \chi_i \) is either an attribute or a constant.

We associate a variable \( z_{t,A_j} \) to each database value \( t[A_j] \), where \( t \) is a tuple in the database instance \( D \) and \( A_j \) is an attribute in \( \mathcal{M}_D \). For every ground substitution \( \theta \) of \( x_1, \ldots, x_k \) such that \( \phi(\theta x_1, \ldots, \theta x_k) \) is true, we will denote as \( T_{\chi_i} \) the set of the tuples involved in the aggregation function \( \chi_i \), that is \( T_{\chi_i} = \{ t : t = \chi_i(\theta y_{i_1}, \ldots, \theta y_{i_m}) \} \).

The translation of \( \chi_i \), denoted as \( \mathcal{P}(\chi_i) \), is defined as follows:
\[
\mathcal{P}(\chi_i) = \left\{ \begin{array}{ll}
\sum_{t \in T_{\chi_i}} z_{t,A_j} & \text{if } e_i = A_j; \\
e_i \cdot |T_{\chi_i}| & \text{if } e_i \text{ is a constant.}
\end{array} \right.
\]
Starting from \( P(\chi_i) \), the whole constraint \( \kappa \) can be expressed as a set \( S \) of linear inequalities as follows. For every ground substitution \( \theta \) of \( x_1, \ldots, x_k \) such that \( \phi(\theta x_1, \ldots, \theta x_k) \) is true, \( S \) contains the following inequality:

\[
\sum_{i=1}^{n} v_i \cdot P(\chi_i) \leq K
\]  

(4)

Observe that this construction is not possible for a non-steady aggregate constraint since, given a database instance \( D \) and an aggregation function \( \chi_i \) in the constraint, we cannot determine \( T_{\chi_i} \): changing a measure value might result in changing the set of the tuples involved the aggregation function.

For the sake of simplicity, in the following we associate to each pair \( \langle t, A_j \rangle \) an integer index \( i \), therefore we write \( z_i \) instead of \( z_{t,A_j} \). If we assume that the number of values involved in constraints in \( AC \) concerning the given database instance \( D \) is \( N \) then the index \( i \) will take values in \([1..N]\).

As shown above, we can translate each steady aggregate constraint into a system of linear inequalities. The translation of all aggregate constraints in \( AC \) produces the system of linear inequalities \( A \cdot Z \leq B \), where \( Z = [z_1, z_2, \ldots, z_N]^T \). This system will be denoted as \( S(AC) \).

In the following we will denote as \( v_i \) the current database value corresponding to the variable \( z_i \). That is, if \( z_i \) is associated with \( t[A_j] \), then \( v_i = t[A_j] \). Every solution \( s \) of \( S(AC) \) corresponds to a (possibly non-minimal) repair \( \rho(s) \) of \( D \) w.r.t. \( AC \). In particular, for each variable \( z_i \) which is assigned by \( s \) a value different from \( v_i \), repair \( \rho(s) \) contains an update assigning the value \( z_i \) to the database item corresponding to \( z_i \).

In order to decide whether a solution \( s \) of \( S(AC) \) corresponds to a card-minimal repair, we must count the number of variables \( s \) which are assigned a value different from the corresponding source value in \( D \). This is accomplished as follows. For each \( i \in [1..N] \), we define a variable \( y_i = z_i - v_i \) on the same domain as \( z_i \). Consider the following system of linear inequalities, which will be denoted as \( S'(AC) \):

\[
\begin{align*}
AZ & \leq B \\
y_i & = z_i - v_i \quad \forall i \in [1..N]
\end{align*}
\]  

(5)

As shown in [15], if a system of equalities has a solution, it has also a solution where each variable takes a value in \([-M, M]\), where \( M \) is a constant equal to \( n \cdot (ma)^{2m+1} \), where \( m \) is the number of equalities, \( n \) is the number of variables and \( a \) is the maximum value among the modules of the system coefficients. It is straightforward to see that \( S'(AC) \) can be translated into a system of linear equalities in augmented form with \( m = N + r \) and \( n = 2 \cdot N + r \), where \( r \) is the number of rows of \( A \).

In order to detect if a variable \( z_i \) is assigned (for each solution of \( S'(AC) \) bounded by \( M \) a value different from the original value \( v_i \) (that is, if \( |y_i| > 0 \)), a new binary variable \( \delta_i \) will be defined. \( \delta_i \) will have value 1 if the value of \( z_i \) differs from \( v_i \), 0 otherwise. To express this condition, we add the following constraints to \( S'(AC) \):

\[
\begin{align*}
y_i & \leq M \delta_i \quad \forall i \in [1..N] \\
-M \delta_i & \leq y_i \quad \forall i \in [1..N] \\
\delta_i & \in \{0, 1\} \quad \forall i \in [1..N]
\end{align*}
\]  

(6)

1 Observe that the size of \( M \) is polynomial in the size of the database, as it is bounded by \( \log n + (2 \cdot m + 1) \cdot \log(ma) \).
The system obtained by assembling $S'(\mathcal{AC})$ with inequalities (6) will be denoted as $S''(\mathcal{AC})$. For each solution $s''$ of $S''(\mathcal{AC})$, the following hold: 1) for each $z_i$ which is assigned in $s''$ a value greater than $v_i$, the variable $\delta_i$ is assigned 1 (this is entailed by constraint $y_i \leq M\delta_i$); 2) for each $z_i$ which is assigned in $s''$ a value less than $v_i$, the variable $\delta_i$ is assigned 1 (this is entailed by constraint $-M\delta_i \leq y_i$). Moreover, for each $z_i$ which is assigned in $s''$ the same value as $v_i$ (that is, $y_i = 0$), variable $\delta_i$ is assigned either 0 or 1.

Obviously each solution of $S''(\mathcal{AC})$ corresponds to exactly one solution for $\mathcal{S}(\mathcal{AC})$ (or, analogously, for $S'(\mathcal{AC})$) with the same values for variables $z_i$, and, vice versa, for each solution of $\mathcal{S}(\mathcal{AC})$ whose variables are bounded by $M$ there is at least one solution of $S''(\mathcal{AC})$ with the same values for variables $z_i$. As solutions of $\mathcal{S}(\mathcal{AC})$ correspond to repairs for $D$, each solution of $S''(\mathcal{AC})$ corresponds to a repair $\rho$ for $D$ w.r.t. $\mathcal{AC}$ such that, for each update $u = (t, A, v)$ in $\rho$ it holds that $|v| \leq M$. Repairs satisfying this property will be said to be $M$-bounded repairs.

In order to consider only the solutions of $S''(\mathcal{AC})$ where each $\delta_i$ is 0 if $y_i = 0$, we consider the following optimization problem $S^*(\mathcal{AC})$, whose goal is minimizing the sum of the values assigned to the variables $\delta_1, \ldots, \delta_N$:

$$\min \sum_{i=1}^{N} \delta_i$$

$$\begin{align*}
AZ &\leq B \\
y_i &= z_i - v_i \quad \forall i \in [1..N] \\
y_i - M\delta_i &\leq 0 \quad \forall i \in [1..N] \\
-y_i - M\delta_i &\leq 0 \quad \forall i \in [1..N] \\
z_i, y_i &\in \mathbb{R} \quad \forall i \in I_R \\
z_i, y_i &\in \mathbb{Z} \quad \forall i \in I_Z \\
\delta_i &\in \{0, 1\} \quad \forall i \in [1..N]
\end{align*}$$

where $I_R \subseteq \{1, \ldots, N\}$ and $I_Z \subseteq \{1, \ldots, N\}$ are the sets of the indexes of the variables $z_1, \ldots, z_N$ (and, equivalently, $y_1, \ldots, y_N$) defined on the domains $\mathbb{R}$ and $\mathbb{Z}$, respectively.

It is straightforward to see that any solution of $S^*(\mathcal{AC})$ corresponds to an $M$-bounded repair $\rho$ for $D$ w.r.t. $\mathcal{AC}$ having minimum cardinality w.r.t. all $M$-bounded repairs for $D$ w.r.t. $\mathcal{AC}$. It can be shown that if a repair exists for $D$ w.r.t. $\mathcal{AC}$, then there is a card-minimal repair $\rho^*$ for $D$ which is $M$-bounded (this follows from Lemma 1 in [11]). This implies that any solution of $S^*(\mathcal{AC})$ corresponds to a card-minimal repair for $D$ w.r.t. $\mathcal{AC}$.

Basically, the minimum value of the objective function of $S^*(\mathcal{AC})$ represents the number of atomic updates performed by any card-minimal repair, whereas the values of variables $z_1, \ldots, z_N, y_1, \ldots, y_N, \delta_1, \ldots, \delta_N$ corresponding to an optimum solution $s^*$ of $S^*(\mathcal{AC})$ define the atomic updates performed by the card-minimal repair $\rho(s^*)$.

6 DART architecture

The DART architecture is shown in Figure 4, where the organization of both the Acquisition and extraction module and the Repairing module of Figure 2 are described in more detail. In the following we discuss the tasks accomplished by these modules.
6.1 Acquisition module
This module performs the task of acquiring the information contained in the (either electronic or paper) input documents, and represents it into an electronic document whose format is suitable for the extraction phase accomplished by the Data Extraction Module. As the current implementation of DART embeds a wrapper working on HTML documents, input documents which are not already in this format are converted into an HTML document by means of a format-conversion tool (in the current implementation this tool supports the conversion of PDF, MSWord, RTF documents). In particular, paper documents are first digitized and processed by means of an OCR tool (yielding PDF documents) whose output is then processed by the converter.

6.2 Data extraction module
The Data extraction module carries out both the information extraction and the database generation tasks. The former task is accomplished by a wrapping sub-module which takes as input the HTML document generated by the Acquisition module as well as a set of extraction metadata providing information on the semantics and the structure of data contained into the input document.

Wrapper
Data to be extracted from the input HTML document are contained into tables whose position inside the document is specified inside the extraction metadata. The information encoded into each table is extracted by evaluating whether its rows match some patterns (namely row patterns) contained in the extraction metadata and defining structure and content of the data to be extracted. The wrapper takes as input a set of row patterns and the HTML document returned by the acquisition module, and returns a set of row pattern instances. A row pattern instance is the result of a matching phase between a table row and the set of row patterns. We point out that, during the matching phase, the wrapper also fixes possible symbol recognition errors in non-numerical strings occurred in the acquisition phase (this is accomplished by comparing acquired strings with terms stored in a dictionary).

Fig. 4. The DART Architecture
Database generator

The Database generator sub-module takes as input the set of row pattern instances returned by the wrapper module and returns a database instance $D$ conforming to the database scheme defined in the extraction metadata.

Extraction metadata specify also classification information providing classification of acquired data depending on the role they play in aggregation constraints. For instance, in Example 1 balance items are classified as detail, aggregate and derived items (the meaning of these classes has been defined in Example 2).

The definition of the database scheme contained in the extraction metadata contains both the definition of the relational scheme (that is, the name of the relations and, for each relation, the names of its attributes) and the correspondence between each relation scheme and the row pattern instances taken as input. For instance in our running example the relational scheme specified in the extraction metadata consists of $\text{CashBudget}(\text{Year}, \text{Section}, \text{Subsection}, \text{Type}, \text{Value})$. Moreover, the extraction metadata contain the specification that attributes Year, Section, Subsection, Value correspond to the row pattern instances, whereas the attribute Type is determined by classification information.

Each row pattern instance taken as input is exploited to insert a new tuple in the corresponding relation. For instance, each tuple $t$ in Figure 3 is obtained from a row pattern instance $r$ returned by the wrapper.

6.3 Repairing module

The input of the repairing module is the database $D$ obtained by the data extraction module and a set $\mathcal{AC}$ of steady aggregate constraints implied by the constraint metadata. The repairing module returns a card-minimal repair for $D$ w.r.t. $\mathcal{AC}$. This is accomplished by means two phases: first, the problem of finding a card-minimal repair for $D$ w.r.t. $\mathcal{AC}$ is translated into an instance of an MILP problem (as we have shown in Section 5), and then such an obtained MILP instance is solved by means of an MILP solver, which is implemented using LINDO API 4.0 (available at www.lindo.com).

Validation Interface

The Validation Interface is the component allowing the operator to interact with DART. When a document is processed, the Validation Interface displays the repair computed by the Repairing module by showing the suggested set of value updates. Then, the operator examines the proposed repair by comparing every updated value with the corresponding source value in the input document. If the operator verifies that the suggested updated values are equal to the corresponding source values, then the repair is accepted and the repaired data is considered as consistent. Otherwise, a new repair is computed by the Repairing module according to operator “instructions”. That is, for each suggested update $u$ which has not been accepted by the operator, the operator can specify the actual source value $v$ corresponding to the database item $d$ changed by $u$. Then an aggregate constraint is added to the set of constraints inputted into the MILP transformer, forcing the value of $d$ to be equal to $v$. Similarly, accepting an update $u$ on the database item $d$ is translated into an aggregate constraint forcing the value of $d$ to be equal to the value suggested by the repair. After this, a new repair is computed, corresponding to the solution of the new MILP instance obtained by assembling the aggregate constraints resulting from Constraint Metadata with those resulting from operator validation. This process goes on until the generated repair is accepted by the operator.
Conclusions and Future work

DART is currently being developed. Both the Acquisition and extraction module and the Repairing module have been implemented, but no user-friendly interface is currently available. Preliminary tests show that DART effectively supports the acquisition of balance data, providing the correct repair of wrongly acquired data in a few iterations in most cases. A more extensive experimental evaluation of system effectiveness will be accomplished on larger data sets when a user-friendly visual interface will be available.

References

On the first-order reducibility of unions of conjunctive queries over inconsistent databases

Domenico Lembo, Riccardo Rosati, and Marco Ruzzi

Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, I-00198 Roma, Italy
{lembo,rosati,ruzzi}@dis.uniroma1.it

Abstract. Recent approaches in the research on inconsistent databases have started analyzing the first-order reducibility of consistent query answering, i.e., the possibility of identifying classes of queries whose consistent answers can be obtained by a first-order (FOL) rewriting of the query, which in turn can be easily formulated in SQL and directly evaluated through any relational DBMS. So far, the investigations in this direction have only concerned subsets of conjunctive queries over databases with key dependencies. In this paper we extend the study of first-order reducibility of consistent query answering under key dependencies to more expressive queries, in particular to unions of conjunctive queries. More specifically: (i) we analyze the applicability of known FOL-rewriting techniques for conjunctive queries in the case of unions of conjunctive queries. It turns out that such techniques are applicable only to a very restricted class of unions of conjunctive queries; (ii) to overcome the above limitations, we define a new rewriting method which is specifically tailored for unions of conjunctive queries. The method can be applied only to unions of conjunctive queries that satisfy an acyclicity condition on unions of conjunctive queries.

1 Introduction

Consistent query answering Research in consistent query answering (CQA) studies the definition (and computation) of “meaningful” answers to queries posed to databases whose data do not satisfy the integrity constraints (ICs) declared on the database schema [2, 11, 4].

Recent studies in this area have established declarative semantic characterizations of consistent query answering over relational databases, decidability and complexity results for consistent query answering, as well as techniques for query processing [2, 6, 11, 4, 3, 5]. In particular, it has been shown that computing consistent answers of conjunctive queries (CQs) is coNP-hard in data complexity, i.e., in the size of the database instance, even in the presence of very restricted forms of ICs (single, unary keys).

From the algorithmic viewpoint, the approach mainly followed is query answering via query rewriting: (i) First, the query that must be processed (usually
a conjunctive query) is reformulated in terms of another, more complex query. Such a reformulation is purely intensional, i.e., the rewritten query is independent of the database instance; (ii) Then, the reformulated query is evaluated over the database instance. Due to the semantic nature and the inherent complexity of consistent query answering, Answer Set Programming (ASP) is usually adopted in the above reformulation step [11, 3, 5], and stable model engines like DLV [13] can be used for query processing.

First-order reducibility of Consistent Query Answering An orthogonal approach to consistent query answering is the one followed by recent theoretical works [2, 6, 10, 12], whose aim is to identify classes of first-order reducible queries, i.e., queries whose consistent answers can be obtained by rewriting the query in terms of a first-order (FOL) query.

The advantage of such an approach is twofold: first, this technique allows for computing consistent answers in time polynomial in data complexity (i.e., for such subclasses of queries, consistent query answering is computationally simpler than for the whole class of CQs); second, consistent query answering in these cases can be performed through standard database technology, since the FOL query synthesized can be easily translated into SQL and then evaluated by any relational DBMS. On the other hand, this approach is only limited to polynomial subclasses of the problem. In particular, Fuxman and Miller in [10] have studied databases with key dependencies, and have identified a broad subclass of CQs that can be treated according to the above strategy.

Since first-order queries can be expressed in SQL, the importance of FOL-reducibility is that, when consistent query answering enjoys this property, we can take advantage of Database Management Systems (DBMS) for answering queries via reformulation into SQL. Notably, in this case, the data complexity of consistent query answering is the one of standard evaluation of FOL queries over databases, i.e., LogSpace.

Our contribution In this paper we study first-order reducibility of consistent query answering for unions of conjunctive queries in the presence of key dependencies. More specifically, our contribution can be summarized as follows:

1. first, we analyze the direct applicability of the rewriting technique of [10] to unions of conjunctive queries. In particular, we characterize the subclass of unions of conjunctive queries for which a first-order rewriting can be computed in a modular way, such that the FOL rewriting of a union of conjunctive queries corresponds to the union of the FOL rewritings of each single conjunctive query. It turns out that this way of FOL-reducing unions of conjunctive queries is possible only for a very restricted class of unions of conjunctive queries;
2. to overcome the limitations of the previous approach, we define a new rewriting method which is specifically tailored for unions of conjunctive queries.

1 We consider here the kernel of the SQL-92 standard, i.e., we see SQL as an implementation of relational algebra.
The method can be applied only to a subclass of unions of conjunctive queries, in particular the queries that satisfy an acyclicity condition on unions of conjunctive queries: for each such query $q$, the method produces a FOL-rewriting of the query whose evaluation produces the consistent answers to $q$.

**Relevance of our results** We believe that the relevance of our study is twofold:

1. Extending the study of first-order reducibility of consistent query answering from conjunctive (i.e., select-project-join) queries to more expressive queries is certainly interesting: in this respect, the extension to unions of conjunctive queries is particularly important, since the possibility of expressing unions is probably the most important expressive feature which is missed by the language of conjunctive queries.

2. As explained in Section 5, we argue that the ability of handling unions of conjunctive queries is necessary in order to extend the first-order reduction techniques of consistent query answering to other forms of integrity constraints, specifically to inclusion dependencies. Besides key dependencies, inclusion dependencies, and in particular foreign keys, are certainly the most important form of integrity constraints in relational schemas; an important outcome of our analysis is that the first-order reduction of conjunctive queries under keys and foreign keys can be reduced to first-order reduction of unions of conjunctive queries under keys. Consequently, the analysis of first-order reduction of unions of conjunctive queries constitutes a necessary first step in order to arrive at the definition of analogous methods for (unions of) conjunctive queries under key and foreign key dependencies.

**Structure of the paper** In the next section, we present some preliminary definitions. In Section 3 we recall the method for first-order reducibility of conjunctive queries under key dependencies and study under which conditions this technique can be directly applied to unions of conjunctive queries. Then, in Section 4 we define a new query rewriting algorithm specifically designed for unions of conjunctive queries, and discuss formal properties of the method. Finally, we conclude in Section 5.

2 Inconsistent databases and consistent answers

**Syntax** We consider to have an infinite, fixed alphabet $\Gamma$ of constants representing real world objects, and we take into account only database instances having $\Gamma$ as domain. Moreover, we assume that different constants in $\Gamma$ denote different objects, i.e., we adopt the so-called unique name assumption.

A database schema $S$ is constituted by a relation signature $\mathcal{A}$, i.e., a set of relation symbols in which each relation is associated with an arity (positive integer) indicating the number of its attributes, and a set of integrity constraints
specified over $\mathcal{A}$. An attribute of a relation symbol $r$ is an integer $b$ such that $1 \leq b \leq n$, where $n$ is the arity of $r$. We consider schemas which contain only key dependencies specified over $\mathcal{A}$. A key dependency (KD) over $\mathcal{A}$ is an expression of the form $\text{key}(r) = \{i_1, \ldots, i_k\}$, where $r$ is a relation symbol of $\mathcal{A}$, and, if $n$ is the arity of $r$, $1 \leq i_j \leq n$ for each $j$ such that $1 \leq j \leq k$. We assume that at most one KD is specified over a relation $r$ and we say that an attribute of $r$ is a key attribute if it belongs to the set $\text{key}(r)$ (otherwise we say that it is a non-key attribute). We denote with the pair $\langle \mathcal{A}, K \rangle$, a database schema $\mathcal{S}$ with signature $\mathcal{A}$ and set of key dependencies $K$ over $\mathcal{A}$.

A term is either a variable or a constant of a (database schema with) signature $\mathcal{A}$ is an expression of the form $p(t_1, \ldots, t_n)$ where $p$ is a relation symbol of arity $n$ and $t_1, \ldots, t_n$ is a sequence of $n$ terms. An atom is called fact if all the terms occurring in it are constants. A database instance $\mathcal{D}$ for $\mathcal{S}$ is a set of facts over $\mathcal{A}$. We denote as $r^\mathcal{D}$ the set $\{ t | r(t) \in \mathcal{D} \}$.

A union of conjunctive queries (UCQ) $q$ of arity $n$ over a (database schema with) signature $\mathcal{A}$ is an expression of the form $h(x_1, \ldots, x_n) := d_1 \vee \ldots \vee d_m$, where the atom $h(x_1, \ldots, x_n)$ is called the head of the query (denoted by $\text{head}(q)$), $d_1 \vee \ldots \vee d_m$ is called the body of the query (denoted by $\text{body}(q)$), and for each $i \in \{1 \ldots m\}$, $d_i$ called the $i$-th disjunct of $q$, is a conjunction of atoms $a_{i_1,1} \land \ldots \land a_{i_k,1}$, whose predicate symbols are in $\mathcal{A}$, such that all the variables occurring in the query head also occur in $d_i$. If $m = 1$, $q$ is simply called conjunctive query (CQ). In a UCQ $q$, we say that a variable is a head variable if it occurs in the query head, while we say that a variable is existential if it only occurs in the query body. Moreover, we call an existential variable shared in a disjunct $d$ of $q$ if it occurs at least twice in $d$ (otherwise we say that it is non-shared in $d$). Obviously, if $q$ is a CQ, an existential variable shared (resp. non-shared) in the unique disjunct of $q$ will be simply called shared (resp. non-shared) in $q$.

A FOL query of arity $n$ is an expression of the form $\{x_1, \ldots, x_n \mid \Phi(x_1, \ldots, x_n)\}$, where $x_1, \ldots, x_n$ are variable symbols and $\Phi$ is a first-order formula with free variables $x_1, \ldots, x_n$.

**Semantics** First, we briefly recall the standard evaluation of queries over a database instance. Let $q$ be the UCQ $h(x_1, \ldots, x_n) := d_1 \vee \ldots \vee d_m$ and let $t = \langle c_1, \ldots, c_n \rangle$ be a tuple of constants of $I$. A set of facts $I$ is an image of $t$ w.r.t. $q$ if there exists a substitution $\sigma$ of the variables occurring in a disjunct $d_i$ of $q$ such that $\sigma(\text{head}(q)) = h(t)$ and $\sigma(d_i) = I$. Given a database instance $\mathcal{D}$, we denote by $q^\mathcal{D}$ the evaluation of $q$ over $\mathcal{D}$, i.e., $q^\mathcal{D}$ is the set of tuples $t$ such that there exists an image $I$ of $t$ w.r.t. $q$ such that $I \subseteq \mathcal{D}$.

Given a FOL query $q$ and a database instance $\mathcal{D}$, we denote by $q^\mathcal{D}$ the evaluation of $q$ over $\mathcal{D}$, i.e., $q^\mathcal{D} = \{(c_1, \ldots, c_n) \mid \mathcal{D} \models \Phi(c_1, \ldots, c_n)\}$, where each $t_j$ is a constant symbol and $\Phi(c_1, \ldots, c_n)$ is the first-order sentence obtained from $\Phi$ by replacing each free variable $x_i$ with the constant $c_i$.

Then, we define the semantics of queries over inconsistent databases. A database instance $\mathcal{D}$ violates the KD $\text{key}(r) = \{i_1, \ldots, i_k\}$ iff there exist two distinct facts $r(c_1, \ldots, c_n), r(d_1, \ldots, d_n)$ in $\mathcal{D}$ such that $c_{ij} = d_{ij}$ for each $j$ such that $1 \leq j \leq k$. 
Let $S = \langle A, K \rangle$ be a database schema. A database instance $D$ is legal for $S$ if $D$ does not violate any KD in $E$.

A set of ground atoms $D'$ is a repair of $D$ under $S$ iff: (i) $D' \subseteq D$; (ii) $D'$ is legal for $S$; (iii) for each $D''$ such that $D' \subset D'' \subseteq D$, $D''$ is not legal for $S$. In words, a repair for $D$ under $S$ is a maximal subset of $D$ that is legal for $S$.

The problem in which we are interested is consistent query answering [2]: given a database schema $S$, a database instance $D$, and a UCQ $q$, return all tuples $t$ of constants of $\Gamma$ such that, for each repair $D'$ of $D$ under $S$, $t \in q_{D'}$. Each such tuple is called consistent answer to $q$ in $D$ under $S$.

Furthermore, analogously to [10], we say that consistent query answering for a class $C$ of UCQs is FOL-reducible (or simply that the class $C$ is FOL-reducible), if for every database schema $S = \langle A, K \rangle$ and every query $q \in C$ over $A$, there exists a FOL query $q_f$ over $A$ such that for every database instance $D$, $t$ is a consistent answer to $q$ in $D$ under $S$ iff $t \in q_{D_f}$. We call such a $q_f$ a FOL-rewriting of $q$ under $S$. Notice that FOL-reducibility is a very interesting property from a practical point of view, since FOL queries correspond to queries expressed in relational algebra (i.e., in SQL). Observe also that every FOL query can be evaluated in LogSpace wrt data complexity, i.e., computational complexity w.r.t. the size of the database instance (see e.g., [1]). It follows that if a class $C$ is FOL-reducible, then consistent query answering for $C$ is in LogSpace wrt data complexity.

3 FOL-rewriting of UCQs via FOL-rewriting of CQs

It is well known that the consistent query answering problem studied in this paper is coNP-hard in data complexity for generic conjunctive queries (and thus for generic unions of conjunctive queries) [4, 6]. As a consequence, the issue of scalability of query answering with respect to (large) database instances turns out to be crucial [3, 8]. In this respect, an interesting approach is the one that aims at identifying subclasses of queries for which the problem is tractable [7, 6], or FOL-reducible [2, 10, 9]. In particular, in [10] the authors study the problem for the class of conjunctive queries, and define a subclass of CQs, called $C_{tree}$, for which they provide an algorithm for FOL-rewriting under schemas which contain only KDs. The class $C_{tree}$ is based on the notion of join graph: a join graph of a conjunctive query $q$ is the graph that contains (i) a node $N_i$ for every atom in the query body, (ii) an arc from $N_i$ to $N_j$ if an existential shared variable occurs in a non-key position in $N_i$ and occurs also in $N_j$, (iii) an arc from $N_i$ to $N_i$ if an existential shared variable occurs at least twice in $N_i$, and at least one occurrence is in a non-key position. According to [10], $C_{tree}$ is the class of conjunctive queries (a) without repeated relation symbols, (b) in which every join from non-key to key attributes involves the entire key of at least one relation and (c) whose join graph is acyclic. As pointed out in [10], this class of queries is very common, since cycles are rarely present in queries used in practice.

A class of CQs slightly more general than $C_{tree}$, called $C_{tree}^+$, has been considered in [12], and a new algorithm, called CQ-FolRewrite, for FOL-rewriting of such CQs has been proposed. Conjunctive queries belonging to such a class...
respect condition (a) and (c) above, but admit also joins from non-key attributes that not necessarily involve the entire key of a relation (i.e., condition (b) above has been removed).2

In what follows we consider the algorithm CQ-FolRewrite and study a possible extension of it in order to deal with queries specified in the more expressive language of unions of conjunctive queries. For the sake of completeness, we show below the algorithm CQ-FolRewrite.

Algorithm CQ-FolRewrite(q, S)
Input: CQ q ∈ C+ CQ tree (whose head variables are x1, . . . , xn);
       schema S = ⟨A, K⟩
Output: FOL query
begin
  compute JG(q);
  return {x1, . . . , xn | ∨ N ∈ roots(JG(q)) NodeRewrite(JG(q), N)}
end

Algorithm NodeRewrite(JG(q), N)
Input: Join Graph JG(q);
       node N of JG(q)
Output: FOL formula
begin
  let a = r(x1/t1, . . . , xn/tn) be the label of N;
  for i := 1 to n do
    if ti ∈ {KB, B} then vi := xi
    else vi := yi, where yi is a new variable
  if each argument of a is of type B or KB then f1 := r(x1, . . . , xn)
  else begin
    let i1, . . . , im be the positions of the arguments of a of type S, U, KB;
    f1 := ∃yi1, . . . , yim. r(v1, . . . , vn)
  end;
  if there exists no argument in a of type B or S then return f1
  else begin
    let p1, . . . , pc be the positions of the arguments of a of type U, S or B;
    let ℓ1, . . . , ℓh be the positions of the arguments of a of type B;
    for i := 1 to c do
      if tp := S then zp := xp else zp := y′p, where y′p is a new variable
    for i := 1 to n do
      if ti ∈ {KB, KU} then wi := vi else wi := zi;
    f2 := ∀zp1, . . . , zpc. r(w1, . . . , wn) → (∨ N′ ∈ jgsucc(N) NodeRewrite(JG(q), N′)) ∧ \[i \in \{ℓ1, . . . , ℓh}\] wi = xi
    return f1 ∧ f2
  end
end

2 The algorithm CQ-FolRewrite takes into account also other forms of integrity constraints specified on the database schema (a.k.a. exclusion dependencies), which are not considered in the present paper.
In the algorithm, we exploit a refined notion of join graph, in which we associate to each node an adornment which specifies the different nature of terms in the atoms, as formally specified below.

**Definition 1.** Let $S = \langle A, K \rangle$ be a database schema, $q$ be a CQ over $A$, and $a = r(x_1, \ldots, x_n)$ be an atom (of arity $n$) occurring in the body of $q$. Then, let $\text{key}(r) = \{i_1, \ldots, i_k\}$ belong to $K$, and let $1 \leq i \leq n$. The type of the $i$-th argument of $a$ in $q$, denoted by $\text{type}(a, i, q)$, is defined as follows:

1. If $i_1 \leq i \leq i_k$, then:
   - if $x_i$ is a head variable of $q$, a constant, or an existential shared variable, then $\text{type}(a, i, q) = \text{KB}$;
   - if $x_i$ is an existential non-shared variable of $q$, then $\text{type}(a, i, q) = \text{KU}$.
2. Otherwise ($i \notin \{i_1, \ldots, i_k\}$):
   - if $x_i$ is a head variable of $q$ or a constant, then $\text{type}(a, i, q) = \text{B}$;
   - if $x_i$ is an existential shared variable of $q$, then $\text{type}(a, i, q) = \text{S}$;
   - if $x_i$ is an existential non-shared variable of $q$, then $\text{type}(a, i, q) = \text{U}$.

Terms typed by $\text{KB}$ or $\text{B}$ are called *bound terms*, otherwise they are called *unbound*. We call the typing of $a$ in $q$ the expression of the form $r(x_1/t_1, \ldots, x_n/t_n)$, where each $t_i$ is the type of the argument $x_i$ in $q$.

In the algorithm, $JG(q)$ denotes the join graph of $q$, in which each node $N_i$ is labelled with the typing of the corresponding atom $a_i$ in $q$, and $\text{jgsucc}(N)$ denotes the set of node which are successors on $N$ in the join graph. Furthermore, $\text{roots}(JG(q))$ denotes the set of nodes that are roots in $JG(q)$ (notice that each join graph for a query of class $C^+_\text{tree}$ is actually a set of trees, i.e. a forest). For a detailed description of the algorithm we refer the reader to [12], where also soundness and completeness of $\text{CQ-FolRewrite}$ with respect to the problem of consistent query answering for CQs belonging to the $C^+_\text{tree}$ class are established.

We are now ready to attack the study of consistent query answering for UCQs specified over database schemas with key dependencies. We start by analyzing the possibility of solving the problem for a UCQ $q$ by simply applying the algorithm $\text{CQ-FolRewrite}$ to each disjunct $d_i$ of $q$, and taking as result the query $q_f$ obtained by the union of the FOL queries produced by each such execution of $\text{CQ-FolRewrite}$. In order to do that, in the following we obviously consider UCQs whose disjuncts are of class $C^+_\text{tree}$. Formally, we provide the following algorithm $\text{UCQ-FolRewrite}$:

**Algorithm UCQ-FolRewrite($q, S$)**

**Input:** UCQ $q = h(x_1, \ldots, x_n) :\text analogue of } d_i \vee \ldots \vee d_m \text{ such that } d_i \in C^+_\text{tree} \text{ for } i \in \{1, \ldots, m\}$; schema $S = \langle A, K \rangle$

**Output:** FOL query

begin
  for $i := 1$ to $m$ do
    begin
      $q_i = h(x_1, \ldots, x_n) :\text analogue of } d_i$
      compute $JG(q_i)$;
    end
Example 1. Consider a database schema \( S = (A, K) \), such that \( A \) contains the binary relation symbols \( r_1, r_2 \) and \( r_3 \), and \( K \) contains the dependencies \( \text{key}(r_1) = \{1\}, \text{key}(r_2) = \{1\}, \text{key}(r_3) = \{1\} \). Consider the UCQ
\[
q := (r_1(x, y) \land r_2(y, z)) \lor (r_3(x, y) \land r_1(y, z))
\]
over \( A \). The join graphs of each disjunct are as follows:
\[
\begin{align*}
r_1(x/KU, y/S) \ (N1) & \quad \rightarrow \quad (N2) \ r_2(y/KB, z/U) \\
r_3(x/KU, y/S) \ (N1) & \quad \rightarrow \quad (N2) \ r_2(y/KB, z/U)
\end{align*}
\]
Now it is easy to see that any disjunct in the query is in class \( C^+_\text{tree} \). Then, the first-order query returned by the execution of \( \text{UCQ-FolRewrite}(q, S) \) is
\[
q_f = \{ | (\exists x, y. r_1(x, y) \land \forall y'. r_1(x, y') \rightarrow \exists z. r_2(y', z)) \lor (\exists x, y. r_3(x, y) \land \forall z. r_3(x, z) \rightarrow \exists z. r_2(y', z)) \}
\]
For such an example it is possible to verify that the query above is actually the FOL-rewriting of the input query \( q \), i.e., for every database instance \( D \), \( t \) is a consistent answer to \( q \) in \( D \) under \( S \) if and only if \( t \in q_f \).

Example 2. Assume to have a database schema \( S = (A, K) \), such that \( A \) contains the relation symbol \( r \) of arity 2, and \( K \) contains the dependency \( \text{key}(r) = \{1\} \). Consider the UCQ \( q := r(x, c_1) \lor r(x', c_2) \over A \), in which \( c_1 \) and \( c_2 \) are different constant symbols. It is immediate to verify that any disjunct in the query is in class \( C^+_\text{tree} \). Then, the first-order query returned by the execution of \( \text{UCQ-FolRewrite}(q, S) \) is
\[
q_f = \{ | (\exists x, y. r(x, y) \land \forall y'. r(x, y') \rightarrow y' = c_1) \lor (\exists x, y. r(x, y) \land (\forall y'. r(x, y') \rightarrow y' = c_2)) \}
\]
Now, assume to have the database instance \( D = \{r(a, c_1), r(a, c_2)\} \), which is not legal for \( S \). It is easy to see that \( D \not\models \Phi \), where \( \Phi \) is the sentence corresponding to the body of \( q_f \), i.e., according to a notation commonly adopted in the database theory for boolean queries, \( {} \not\in D \), where \( {} \) indicates the empty tuple. On the other hand, the repairs of \( D \) under \( S \) are \( \mathcal{R}_1 = \{r(a, c_1)\} \) and \( \mathcal{R}_2 = \{r(a, c_2)\} \), and the body of the query \( q \) evaluates to true in both \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \), i.e. \( {} \) is a consistent answer to \( q \) in \( D \) under \( S \).
The example above shows that the algorithm UCQ-FolRewrite is in general incomplete (even if it is easy to see that it is always sound). This is mainly due to the fact that separately rewriting single disjuncts does not take into account the interaction that may exist between them. Indeed, the body of the FOL-rewriting that the algorithm constructs for each single disjunct \( d_i \) (i.e., \( \bigwedge_{N \in \text{roots}(\text{JG}(q_i))} \text{NodeRewrite}(\text{JG}(q_i), N) \)) is a FOL formula, which we denote with \( \phi \), such that, given an assignment of the free variables of \( \phi \) (i.e., a tuple of constants \( t \)), the sentence \( \phi(t) \) is satisfied only by those database instances \( D \) such that in any repair of \( D \) there is an image of \( t \) w.r.t the disjunct \( d_i \).

On the other hand, for a union of conjunctive queries \( q \), for a tuple \( t \) to be a consistent answer to \( q \) it is sufficient that in any repair of \( D \) there exists an image of the tuple w.r.t. \( q \), i.e. with respect to any disjunct \( d_j \) of \( q \) (in other words, the disjunct which provides the image has not to be the same in any repair). This is actually the case we have in Example 2.

Despite the above limitations of the algorithm, we are able to identify a subclass of conjunctive queries for which the algorithm UCQ-FolRewrite is sound and complete. To this aim, we provide the following definition.

**Definition 2.** Let \( S = \langle A, K \rangle \) be a database schema, let \( r \) be a relation symbol of \( A \) such that \( \text{key}(r) = \{1, \ldots, n\} \in K \). Let \( q \) be a UCQ over \( A \) and let \( a_1 = r(a, c_1) \) and \( a_2 = r(a, c_2) \) be two different atoms occurring respectively in two different disjuncts \( d_1 \) and \( d_2 \) of \( q \), such that \( x = x_1, \ldots, x_n, y = y_1, \ldots, y_m, z = x_1, \ldots, x_n, w = w_1, \ldots, w_m \) are sequences of terms. Then, we say that \( a_1 \) and \( a_2 \) are interacting in \( q \) if

1. the sequences of terms in key position in \( a_1 \) and \( a_2 \) unify, i.e., there exists a unifier between \( x \) and \( z \);
2. there exists \( j \in \{1, \ldots, m\} \) such that \( y_j \) and \( w_j \) are not identical constants;
3. \( a_1 \) (resp. \( a_2 \)) is such that either \( a_1 \) is not a leaf in the join graph of \( d_1 \) (resp. \( d_2 \)) or there exists a non-key argument of \( a_1 \) (resp. \( a_2 \)) which is bound.

Notice that the atoms \( r(a, c_1) \) and \( r(a, c_2) \) in the example above are interacting atoms in the query \( q \). Now we are able to define the class we were looking for.

**Definition 3.** A UCQ \( q \) belongs to the class \( \text{UCQ}_N^I \) of non-interacting UCQs if:

- each disjunct \( d_i \) of \( q \) is in \( \text{C}^+_\text{tree} \);
- there do not exist two interacting atoms \( a_1 \) and \( a_2 \) in \( q \).

It is easy to see that the query in Example 1 belongs to the class \( \text{UCQ}_N^I \), whereas the query in in Example 2 does not.

**Theorem 1.** Let \( S = \langle A, K \rangle \) be a database schema, \( q \in \text{UCQ}_N^I \) be a query over \( S \). Then, the FOL query \( q_f \) returned by the algorithm UCQ-FolRewrite\((q, S)\) is a FOL-rewriting of \( q \) under \( S \).

In other words, the above theorem states that the problem of consistent query answering under key dependencies is FOL-reducible for the class \( \text{UCQ}_N^I \).
4 Algorithm

In this section we try to overcome the limitations of the rewriting technique presented in the previous section, by defining a new FOL-rewriting algorithm for UCQs. Based on such an algorithm, we are able to identify the class of acyclically interacting queries, a class of UCQs which extends the class UCQ_{NI} defined in Section 3, and to prove that acyclically interacting queries constitute a class of FOL-reducible queries under key dependencies.

4.1 The algorithm UCQ-FolRewriteNew

We are now ready to define the algorithm UCQ-FolRewriteNew, a FOL rewriting algorithm for UCQs that, differently from the previous algorithm UCQ-FolRewrite, takes into account the semantic interactions between the query disjuncts.

In the algorithm UCQ-FolRewriteNew (and in the other algorithms iteratively invoked by UCQ-FolRewriteNew and presented in this section), with a little abuse of terminology we call typed query associated to a query \( q \) the query \( q^t \) obtained from \( q \) by replacing each atom with its typing. Analogously, the typed disjunct associated to a disjunct \( d \) is the disjunct \( d^t \) obtained from \( d \) by replacing each atom with its typing. Coherently to the above definitions, when the operator \( JG \), used for constructing the join graph of a query, is applied to a typed query \( q^t \), the nodes of \( JG(q^t) \) respect the typing specified by \( q^t \), i.e., each node of the graph is labeled with the corresponding typing indicated in \( q^t \). We also point out that in the query \( Q \) in input to UCQ-FolRewriteNew, each variable symbol only occurs in a single disjunct of \( Q \), and the new variables introduced by the algorithm NodeRewriteNew (see below) are always fresh symbols with respect to all the executions of the algorithm.

The algorithm UCQ-FolRewriteNew is the following:

**Algorithm UCQ-FolRewriteNew**

**Input:** a first-order reducible UCQ \( Q = h(x_1, \ldots, x_n) :– d_1 \lor \ldots \lor d_m \); schema \( S = \langle A, K \rangle \)

**Output:** FOL query (representing the rewriting of \( Q \))

begin
  let \( Q^t \) be the typed query associated to \( Q \);
  for \( i := 1 \) to \( m \) do
    let \( d^t_i \) be the typed disjunct associated to \( d_i \);
    return \( \{ x_1, \ldots, x_n \mid \bigvee_{i=1, \ldots, m} \text{DisjunctRewrite}(d^t_i, Q^t, S, (\emptyset, \emptyset)) \} \)
  end

end

The above algorithm calls the following algorithm DisjunctRewrite, which computes the rewriting of a single disjunct \( d_i \) of the UCQ:

**Algorithm DisjunctRewrite**

**Input:**
a typed union of conjunctive queries $Q = h(x_1, \ldots, x_n) :- d_1 \lor \ldots \lor d_m$;
a typed disjunct $d$ that appears in $Q$;
schema $S = (A, K)$;
$P = (M, E)$ where $M$ is a list of atoms and $E$ is a set of equalities;
**Output:** FOL query (representing the rewriting of the disjunct $d$)

\[
\begin{aligned}
q &= h(x_1, \ldots, x_n) :- d; \\
\text{compute } JG(q); \\
\text{return } \{ \bigwedge_{N \in \text{roots}(JG(q))} \text{NodeRewriteNew}(JG(q), N, Q, S) \}
\end{aligned}
\]

In turn, DisjunctRewrite calls the following algorithm NodeRewriteNew:

**Algorithm NodeRewriteNew** ($JG(q), N, Q, S, P$)

**Input:** join graph $JG(q)$;
node $N$ of $JG(q)$;
a typed query $Q = h(x_1, \ldots, x_n) :- d_1 \lor \ldots \lor d_m$;
schema $S = (A, K)$;
$P = (M, E)$ where $M$ is a list of atoms and $E$ is a set of equalities;
**Output:** FOL formula

\[
\begin{aligned}
\text{let } a &= r(x_1/t_1, \ldots, x_n/t_n) \text{ be the label of } N; \\
\text{for } i := 1 \text{ to } n \text{ do } \\
\text{if } t_i \in \{KB, B\} \text{ then } v_i := x_i; \\
\text{else } v_i := y_i, \text{ where } y_i \text{ is a new variable; } \\
\text{if each argument of } a \text{ is of type } B \text{ or } KB \text{ then } f_1 := r(x_1, \ldots, x_n) \\
\text{else begin } \\
\text{let } i_1, \ldots, i_m \text{ be the positions of the arguments of } a \text{ of type } S, U, KU; \\
f_1 := \exists y_{i_1}, \ldots, y_{i_m}. r(v_1, \ldots, v_n) \\
\text{end; } \\
\text{if there exists no argument in } a \text{ of type } B \text{ or } S \text{ then return } f_1 \\
\text{else begin } \\
\text{let } p_1, \ldots, p_c \text{ be the positions of the arguments of } a \text{ of type } U, S \text{ or } B; \\
\text{let } e_1, \ldots, e_h \text{ be the positions of the arguments of } a \text{ of type } B; \\
\text{for } i := 1 \text{ to } c \text{ do } \\
\text{if } t_{p_i} = S \text{ then } z_{p_i} := x_{p_i} \text{ else } z_{p_i} := y''_i, \text{ where } y''_i \text{ is a new variable } \\
\text{for } i := 1 \text{ to } n \text{ do } \\
\text{if } t_i \in \{KB, KU\} \text{ then } w_i := v_i \text{ else } w_i := z_i; \\
\text{if } \text{occurs}(H_{kd}(a), P) \text{ then } f_2 = \left( \bigwedge_{N' \in \text{jgsucc}(N)} \text{NodeRewriteNew}(JG(q), N', Q, S, P) \land \bigwedge_{i \in \{e_1, \ldots, e_h\}} w_i = x_i \right) \\
\text{else begin } \\
M = M \cup \{H_{kd}(a)\}; \\
E = E \cup \{w_1 = u_1, \ldots, w_n = u_n\}.
\end{aligned}
\]
Consider again the query

The function call or (in the disjunct occurring in such atom; different from the disjunct which contain atoms interacting with the atom corresponding to the node of terms contained in \( M \) list atom \( a \) avoids useless calls of the algorithm \( \text{NodeRewriteNew} \). Furthermore, \( r(c') \) (i.e., such that \( r(c) \) and \( r(c') \) have the same key) that belongs to an image of \( t \) w.r.t. a disjunct \( d_j \) of \( Q \), might not be part of any image of the same disjunct \( d_j \) but may be part of an image of \( t \) w.r.t. another disjunct \( d_j \) of the query. Thus, the formula in the FOL-rewriting must look for the existence of such an image of \( d_j \). It can be shown that this non-local check must be performed only in the presence of interacting atoms in \( Q \). More precisely, when \( \text{NodeRewriteNew} \) is computing the rewriting of a node corresponding to an atom \( a \), it must recursively invoke \( \text{DisjunctRewrite} \) only for each disjunct \( d_j \) such that there is an atom \( b \) in \( d_j \) that is interacting with \( a \) in \( Q \).

In the algorithms \( \text{DisjunctRewrite} \) and \( \text{NodeRewriteNew} \), \( \mathcal{P} \) is the pair \((M, E)\) in which \( M \) is a list of atoms and \( E \) is a set of equalities of terms. Each atom in the list \( M \) is obtained by means of the operator \( \Pi_{kd} \) applied to a typing \( a = r(x_1/t_1, \ldots, x_n/t_n) \) (i.e., a label of a node of a join graph). \( \Pi_{kd}(a) \) returns the atom \( r(x_{i_1}, \ldots, x_{i_a}) \), where \( i_1, \ldots, i_a \) are positions of the arguments of \( a \) of type \( KU \) or \( KB \), i.e., \( x_{i_1}, \ldots, x_{i_a} \) are the key-arguments of the atom \( r(x_1, \ldots, x_n) \). The function call occurs(\( \Pi_{kd}(a), \mathcal{P} \)) returns true if the atom \( \Pi_{kd}(a) \) is in the list \( M \) or it can be constructed from an atom of \( M \) according to the equalities of terms contained in \( E \). Otherwise occurs(\( \Pi_{kd}(a), \mathcal{P} \)) returns false. This check avoids useless calls of the algorithm \( \text{DisjunctRewrite} \) and guarantees termination of the procedure. Furthermore, \( \text{Int}(N) \) denotes the set of disjuncts of \( Q \) that contain atoms interacting with the atom corresponding to the node \( N \) (and different from the disjunct which \( N \) belongs to); \( u_1, \ldots, u_n \) denote the terms occurring the interacting atom in the disjunct \( d_j \), and \( u_{j_1}, \ldots, u_{j_j} \) are the variables occurring in such atom; \( \tau \) is an operator which modifies the typing of each atom (in the disjunct \( d_j \) and in the query \( Q \) in the two invocations \( \tau(d_j) \) and \( \tau(Q) \), respectively) by assigning \( KB \) to the key arguments of the interacting atom, and \( B \) to the other (non-key) arguments.

**Example 3.** Consider again the query \( q \) of Example 2, and execute the algorithm \( \text{UCQ-FolRewriteNew}(q, S) \). Then the FOL-rewriting produced by the algorithm
is as follows

\[
\{ \exists y_1. r(y_1, c_1) \land \forall y_2. r(y_2, y_2) \rightarrow y_2 = a \lor (\exists y_3. y_3 = a \land r(y_3, c_2)) \land (\exists y_4. y_4 = a \land \forall y_5. r(y_5, y_5) \rightarrow y_5 = a \land r(y_5, c_3)) \}\.
\]

Notice that in such a case the check on the execution of DisjunctRewrite, which we have talked about above, avoids the execution of identical calls of such a procedure.

We finally point out that the rewriting produced by the algorithm can be refined in order to get a simplified version of it (which could be evaluated in a more efficient way). However, this is outside the scope of the present paper.

### 4.2 Termination and correctness

The algorithm UCQ-FolRewriteNew in general does not terminate. In order to characterize the class of queries for which the algorithm terminates, we give the following definitions.

**Definition 4.** Given two atoms \( a = r(x_1, \ldots, x_n, w_1, \ldots, w_m), \ b = r(y_1, \ldots, y_n, z_1, \ldots, z_m) \), where \( \text{key}(r) = \{1, \ldots, n\} \), we say that \( a \) and \( b \) are key-unifiable if, for each \( i \) s.t. \( 1 \leq i \leq n \): (i) \( x_i \) is a variable; or (ii) \( y_i \) is a variable; or (iii) \( x_i = y_i \). If \( a \) and \( b \) are key-unifiable, we denote by \( \sigma_{a\rightarrow b} \) the substitution \( \{ y_i \leftarrow x_i \mid 1 \leq i \leq n \text{ and } y_i \text{ is a variable}\}. \)

**Definition 5.** A UCQ \( Q = \{x_1, \ldots, x_m \mid d_1 \lor \ldots \lor d_n\} \) has a \( \lor \)-cycle if there exists a sequence \( d_{i_1}, \ldots, d_{i_k} \) (with \( k > 1 \)) and a sequence of relation symbols \( r_{j_1}, \ldots, r_{j_{k-1}} \) such that:

- \( d_{i_1} = d_{i_{k}}; \)
- \( i_k = i_1; \)
- for each \( h \) s.t. \( 1 \leq h \leq k-1 \), \( r_{j_h} \) occurs both in \( d_{i_{h+1}} \) and in \( d_{i_{h-1}}; \)
- let \( a \) be the atom with relation \( r_{j_h} \) in \( d_{i_{h}} \) and let \( b \) be the atom with relation \( r_{j_k} \) in \( d_{i_{k-1}} \). Then, \( a \) and \( b \) are key-unifiable. Moreover, for each \( h \) s.t. \( 1 \leq h \leq k-1 \), \( d_{i_{h+1}} = \sigma_{a\rightarrow b}(d_{i_{h-1}}); \)
- the key arguments of \( r_{j_1}(d_{i_{h}}) \) contain at least one existential variable not occurring in the key arguments of \( r_{j_1}(d_{i_{1}}) \).

In a \( \lor \)-cycle, the disjuncts \( d_{i_j} \) and \( d_{i_{j+1}} \) are connected through two atoms \( a \) (occurring in \( d_{i_j} \)) and \( b \) (occurring in \( d_{i_{j+1}} \)) such that \( a \) and \( b \) are on the same relation symbol \( r \). The key arguments of \( a \) are “passed” to \( b \), thus \( d_{i_{j+1}} \) is transformed according to such a substitution.

It is easy to verify that a a necessary condition for a UCQ \( Q \) to have a \( \lor \)-cycle is the presence of interacting atoms in \( Q \).

**Definition 6.** A UCQ \( Q = \{x_1, \ldots, x_m \mid d_1 \lor \ldots \lor d_n\} \) belongs to the class \( UCQ_A \) of acyclically interacting UCQs if:

\(^3\)In the definition of key-unifiable atoms, head variables are considered as constants.
1. each conjunction $d_i$ is such that the CQ $\{x_1, \ldots, x_m \mid d_i\}$ belongs to $C^+_\text{tree}$;
2. there are no $\lor$-cycles in $Q$.

Informally, according to the above definition, a UCQ $Q$ is acyclically interacting if the interacting atoms in the query disjuncts are such that they do not constitute a $\lor$-cycle in $Q$, i.e., a cycle of interactions that, starting from an atom $r(x)$, cycles back to the same atom introducing at least one existential variable in the key arguments of the atom.

From the semantic viewpoint, this corresponds to the fact that, when checking for the opponents of an image $r(t)$ of $a$, we need to check for the opponents of another image $r(t')$ of $a$, where $t'$ has a new value that does not occur neither in $t$ nor in the query $Q$. This immediately implies non-termination of the algorithm UCQ-FolRewriteNew, since at every such iteration there are new key arguments in the call to NodeRewriteNew for the atom $a$. Vice versa, the absence of such a cycle implies termination of the algorithm, since no new term (with respect to the terms occurring in the query $Q$) is introduced in the calls to NodeRewriteNew, hence the number of possible instantiations of the calls to NodeRewriteNew is finite.

The following property formalizes the fact that the class of acyclically interacting queries is precisely the class of UCQs for which the algorithm UCQ-FolRewriteNew terminates.

**Theorem 2.** Let $Q$ be a UCQ, and let $S$ be a schema. The execution of the algorithm UCQ-FolRewriteNew with input $Q$ terminates if and only if $Q$ is acyclically interacting.

Moreover, the following theorem establishes soundness and completeness of the algorithm UCQ-FolRewriteNew for the class of acyclically interacting UCQs.

**Theorem 3.** If $Q$ is acyclically interacting, then for every database instance $D$ for $S$, a tuple $t$ is a consistent answer to $Q$ in $D$ under $S$ iff $t \in Q^D_r$, where $Q_r$ is the FOL query returned by UCQ-FolRewriteNew($Q$) (i.e., the FOL query returned by the algorithm UCQ-FolRewrite($Q$) is a FOL-rewriting of $q$ under $S$).

As a corollary of the above theorem, we obtain that the class UCQ$_{AI}$ is FOL-reducible.

Finally, we point out that:
- the class of UCQ$_{AI}$ is a proper superset of the class of UCQs UCQ$_{NI}$ analyzed in Section 3 and for which the algorithm UCQ-FolRewrite is complete;
- if $Q$ is a query in the class UCQ$_{NI}$, the algorithms UCQ-FolRewrite($Q$) and UCQ-FolRewriteNew($Q$) return exactly the same FOL query.

5 Discussion and Conclusions

We believe that the study of first-order reducibility of consistent query answering for unions of conjunctive queries is relevant per se, since the possibility of expressing unions in queries is an important feature which has practical relevance.
However, we argue that the ability of handling unions of conjunctive queries is necessary in order to solve via FOL-rewriting techniques the problem of consistent query answering for (unions of) conjunctive queries issued over database schemas which contain keys and foreign keys under the loosely-sound semantics, a repair semantics which allows for properly dealing with both incomplete and inconsistent databases\footnote{Notice that also inclusion dependencies of a particular form which guarantees decidability of the consistent query answering problem might be considered \cite{4}.} \cite{4,5}.

Formally, given a database schema $S$ which contains keys and foreign keys, a loosely-sound repair of a database $D$ is any database legal for $S$ that contains a repair (as so far intended in the present paper and formally specified in Section 2) of $D$ under $S'$, where $S'$ is obtained from $S$ disregarding foreign key dependencies. Roughly speaking, such semantics adds the ability to deal with inconsistent databases to the first-order semantics commonly adopted for dealing with incomplete databases. Indeed, in intuitive terms, it maintains the ability of the first-order semantics to deliberately add facts to a database instance (property that can be exploited to satisfy those dependencies that can be satisfied by adding facts, as foreign keys), but it also allows for a (minimal) deletion of facts, thus enabling the repairing of database instances with respect to those dependencies, as key dependencies, that may generate inconsistency according to the first-order semantics.

Notably, as showed in \cite{5}, in order to solve consistent query answering for (unions of) conjunctive queries under the loosely-sound semantics, it is possible to separately dealing with keys and foreign keys. According to the procedure provided in \cite{5}, a query $q$ is first processed only according to the foreign keys issued over the database schema. Such a pre-processing produces a union of conjunctive queries $Q$. Then, it is sufficient to solve consistent query answering for the UCQ $Q$ over the same database schema in which foreign keys have been dropped. It is immediate to see, that if the query $Q$ obtained is of class $UCQ_{AI}$, then to solve the second problem we can apply the algorithm UCQ-FolRewrite presented in this paper.

Consequently, even if still preliminary, the analysis of first-order reduction of unions of conjunctive queries we have presented turns out to be a necessary first step in order to arrive to the definition of analogous methods for (unions of) conjunctive queries under key and foreign key dependencies.

References


Semantically Correct Query Answers in the Presence of Null Values

Loreto Bravo and Leopoldo Bertossi
Carleton University
School of Computer Science
Ottawa, Canada.
{lb Bravo, bertossi}@cs.carleton.ca

Abstract. For several reasons a database may not satisfy a given set of integrity constraints (ICs), but most likely most of the information in it is still consistent with those ICs, and could be retrieved when queries are answered. Consistent answers wrt a set of ICs have been characterized as answers that can be obtained from every possible consistent but minimally repaired version of the original database. In this paper we show and analyze how to specify the repairs of a database that contains null values using disjunctive logic programs with stable model semantics. We analyze in detail the presence of referential ICs, for which the repairs are obtained by introduction of null values that do not propagate through other constraints. For this purpose, we propose first a precise semantics for IC satisfaction in a database with null values that is compatible with and an abstraction of the way null values are treated in commercial database management systems. We also introduce and analyze logic programs that specify the database repairs.

1 Introduction

In databases, integrity constraints (ICs) capture the semantics of the application domain, and help maintain the correspondence between this domain and the database when updates are performed. However, there are several reasons for a database to be or become inconsistent wrt a given set of ICs [6]; and sometimes it could be difficult, impossible or undesirable to repair the database in order to restore consistency [6]. This process might be too expensive; useful data might be lost; it may not be clear how to restore the consistency, and sometimes even impossible, e.g. in virtual data integration, where the access to the autonomous data sources may be restricted [8].

In those situations, possibly most of the data is still consistent and can be retrieved when queries are posed to the database. In [2] consistent data is characterized as the data that is invariant under certain minimal forms of restoration of consistency, i.e. as the data that is present in all minimally repaired and consistent versions of the original instance (the repairs). In particular, an answer to a query is defined as consistent when it can be obtained as a standard answer to the query from every possible repair.

More precisely, a repair of a database instance \( D \), as introduced in [2], is a new instance of the same schema as \( D \) that satisfies the given ICs, and makes minimal under set inclusion the symmetric set difference with the original instance, taken both instances as sets of ground database atoms.

In [2, 12, 14, 18] algorithms and implementations for consistent query answering (CQA) have been presented, i.e. for retrieving consistent answers from inconsistent databases. All of them work only with the original, inconsistent database, without restoring its consistency. That is, inconsistencies are solved at query time. This is in correspondence with the idea that the above mentioned repairs provide an auxiliary concept for defining the right semantics for consistent query answers. However, those algorithms apply to restricted classes of queries and constraints, basically those for which the intrinsic complexity of CQA is still manageable [13].
In [3, 21, 5, 6] a different approach is taken: database repairs are specified as the stable models of disjunctive logic programs, and in consequence consistent query answering amounts to doing cautious or certain reasoning from logic programs under the stable model semantics. In this way, it was possible to handle any set of universal ICs and any first-order query, and even beyond that, e.g. queries expressed in extensions of Datalog. It is important to realize that the data complexity of query evaluation in disjunctive logic programs with stable model semantics [16] matches the intrinsic data complexity of CQA [13], namely both of them are \( \Pi^2_P \)-complete.

In [9] the methodology presented in [5, 6], based on specifying repairs using logic programs with extra annotation constants, was systematically extended in order to handle both; (a) databases containing null values, and (b) referential integrity constraints (RICs) whose satisfaction is restored via introduction of null values. Those introduced null values do not propagate through other ICs, which required a semantics for integrity constraints satisfaction under null values that accepts that tuples with null values (even in attributes that are not relevant to check the IC) are assumed to not generate any inconsistencies. This strategy to deal with referential ICs was motivated by results presented in [10], where it was shown that repairing RICs introducing arbitrary values from the underlying database domain easily leads to the undecidability of CQA.

The approach in [9] has two shortcomings. First, the repairs are all correctly captured as stable models of the program, but in some cases there are more stable models than repairs, and, secondly, the assumption is made that the set of RICs is acyclic. Here, we extend the approach and results in [9] in several ways. First, we give a new semantics for integrity constraint satisfaction in the presence of null values that is both sensitive to the relevance of the kind of occurrence of a null value in a relation, and also compatible with the way null values are usually treated in database management systems. Secondly, the repair programs are modified in such a way that the expected one-to-one correspondence between the stable models and repairs is recovered, even for cyclic sets of RICs. In doing this, we obtain the decidability of CQA for any set of ICs and queries. Finally we study classes of ICs for which the specification can be optimized and a lower complexity for CQA can be obtained.

2 Preliminaries

We concentrate on relational databases, and we assume we have a fixed relational schema \( \Sigma = (U, R, B) \), where \( U \) is the possibly infinite database domain such that \( null \in U \), \( R \) is a fixed set of database predicates, and \( B \) is a fixed set of built-in predicates, like comparison predicates. The schema determines a language \( \mathcal{L}(\Sigma) \) of first-order predicate logic. A database instance \( D \) compatible with \( \Sigma \) can be seen as a finite collection of ground atoms of the form \( P(c_1, ..., c_n) \),\(^1\) where \( P \) is a predicate in \( R \) and \( c_1, ..., c_n \) are constants in \( U \). Built-in predicates have a fixed and equal extension in every database instance, not subject to any changes. We need to define ICs because their syntax is fundamental for what follows.

An integrity constraint is a sentence in \( \mathcal{L}(\Sigma) \) of the form:

\[
\forall \bar{x} \left( \bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \exists \bar{y} \bigvee_{j=1}^{n} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi \right),
\]

\(1\)

where \( P_i, Q_j \in R \), \( \varphi \) is a disjunction of built-in atoms from \( B \), \( \bar{x} = \bigcup_{i=1}^{m} \bar{x}_i \), \( \bigcup_{i=1}^{n} \bar{y}_i \) \( \subseteq \{ \bigcup_{i=1}^{m} \bar{x}_i \} \), and \( \{ \bigcup_{i=1}^{m} \bar{x}_i \} \cap \{ \bigcup_{j=1}^{n} \bar{z}_j \} = \emptyset \). The antecedent of an integrity constraint is \( \bigwedge_{i=1}^{m} P_i(\bar{x}_i) \) and the consequent is \( \bigvee_{j=1}^{n} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi \)

\(1\) Also called database tuples. Finite sequences of constants in \( U \) are simply called tuples.
A universal integrity constraint (UIC) has the form (1), but with $\bigcup_{j=1}^{m} \bar{x}_j = \emptyset$, i.e. without existentially quantified variables:

$$\forall \bar{x} \left( \bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi \right),$$

(2)

We will assume that there is a propositional atom $\text{false} \in B$ that is always false in a database. Then, a denial constraint can be expressed as: $\forall \bar{x} \left( \bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \text{false} \right)$. A referential integrity constraint (RIC) is of the form (1), but with $m = n = 1$ and $\varphi = \emptyset$, i.e. of the form$^2$: (here $\bar{x}' \subseteq \bar{x}$ and $P, Q \in \mathcal{R}$)

$$\forall \bar{x} \left( P(\bar{x}) \rightarrow \exists \bar{y} \; Q(\bar{x}', \bar{y}) \right).$$

(3)

When writing ICs, we will usually leave the prefix of universal quantifiers implicit. This class of ICs includes those most commonly found in the database practice, e.g. denial constraints, functional dependencies, and both partial and full inclusion dependencies (INDs) [1]. Partial INDs contain existential quantifiers, but not the full INDs, that are then universal constraints. We can also specify check constraints, that allow to express conditions on each row in a table, so they can be formulated with one predicate in the antecedent of (1) and only a formula $\varphi$ in the consequent. For example, $\forall xy \left( P(x, y) \rightarrow y > 0 \right)$ is a check constraint. In the following we will assume that we have a fixed set $\mathcal{IC}$ of ICs of the form (1) that is logically and finitely consistent, in the sense that it is satisfied in some database instance compatible with schema $\Sigma$.

**Example 1.** For $\mathcal{R} = \{P, R, S\}$ and $B = \{>, =, \text{false}\}$, the following are ICs: (a) $\forall xyzw \left( P(x, y) \land R(y, z, w) \rightarrow S(x) \lor (z \neq 2 \lor w \leq y) \right)$ (universal). (b) $\forall xy (P(x, y) \rightarrow \exists z \; R(x, y, z))$ (referential). (c) $\forall xy (S(x) \rightarrow \exists yz (R(x, y) \lor R(x, y, z)))$. □

Notice that defining $\varphi$ in (1) as a disjunction of built-in atoms is not an important restriction, because an IC that has $\varphi$ as a more complex formula can be transformed into a set of constraints of form (1). For example, the formula $\forall xy \left( P(x, y) \rightarrow (x > y \lor (x = 3 \land y = 8)) \right)$ can be transformed into: $\forall xy \left( P(x, y) \rightarrow (x > y \lor x = 3) \right)$ and $\forall xy \left( P(x, y) \rightarrow (x > y \lor y = 8) \right)$.

The dependency graph $G(\mathcal{IC})$ [11] for a set of ICs $\mathcal{IC}$ of the form (1) is defined as follows: Each database predicate $P$ in $\mathcal{R}$ appearing in $\mathcal{IC}$ is a vertex, and there is a directed edge $(P_i, P_j)$ from $P_i$ to $P_j$ iff there exists a constraint $ic \in \mathcal{IC}$ such that $P_i$ appears in the antecedent of $ic$ and $P_j$ appears in the consequent of $ic$.

**Example 2.** For the set $\mathcal{IC}$ containing the UICs $ic_1: S(x) \rightarrow Q(x)$ and $ic_2: Q(x) \rightarrow R(x)$, and the RIC $ic_3: Q(x) \rightarrow T(x, y)$, the following is the dependency graph $G(\mathcal{IC})$:

![Dependency Graph](image)

We have labelled the edges just for reference. Edges 1 and 2 correspond to the constraints $ic_1$ and $ic_2$, resp., and edge 3 to $ic_3$. □

A connected component in a graph is a maximal subgraph such that for every pair $(A, B)$ of its vertices, there is a path from $A$ to $B$ or from $B$ to $A$. For a graph $G$, $\mathcal{C}(G) := \{ c \mid c \text{ is a connected component in } G \}$; and $\mathcal{V}(G)$ is the set of vertices of $G$.

---

$^2$ To simplify the presentation, we are assuming the existential variables appear in the last attributes of $Q$, but they may appear anywhere else in $Q$. 

35
Definition 1. Given a set $IC$ of UICs and RICs, $IC_U$ denotes the set of UICs in $IC$. The contracted dependency graph, $G^C(IC)$, of $IC$ is obtained from $G(IC)$ by replacing for every $c \in C(G(IC_U))$ the vertices in $V(c)$ by a single vertex and deleting all the edges associated to the elements of $IC_U$. Finally, $IC$ is said to be RIC-acyclic if $G^C(IC)$ has no cycles.

Example 3. (example 2 cont.) The contracted dependency graph $G^c(IC)$ is obtained by replacing in $G(IC)$ the edges 1 and 2 and their end vertices by a vertex labelled with $\{Q, R, S\}$. Since there are no loops in $G^c(IC)$, the set $IC$ is RIC-acyclic.

If we add a new UIC: $T(x,y) \rightarrow R(y)$ to $IC$, all the vertices belong to the same connected component. $G(IC)$ and $G^c(IC)$ are, respectively:

Since there is a self-loop in $G^c(IC)$, the set of ICs is not RIC-acyclic.

3 IC Satisfaction in Incomplete Databases

Here we refer to incomplete databases in the classic sense that there is incomplete information that is represented by some sort of null values [22] (cf. also [20]). More recently, the notion of incomplete database has been used in the context of virtual data integration [24, 8], referring to data sources that contain a subset of the data of its kind in the global system; and in inconsistent databases [10, 15], referring to the fact that inconsistencies may have occurred due to missing information and then, repairs are obtained through insertion of new tuples.

There is no agreement in the literature on the semantics of null values in relational databases. There are several different proposals in the research literature [32, 4, 27, 29], in the SQL standard [34, 23], but also implicit semantics in the different ways null values are handled in commercial database management systems (DBMSs).

Not even within the SQL standard there is an homogenous and global semantics of integrity constraint satisfaction in databases with null values; rather different definitions of satisfaction are given for each type of constraint, and in the case of foreign key constraints, three different semantics are suggested. Commercial DBMSs implement only the so-called simple semantics for foreign key constraints.

One of the reasons why it is difficult to agree on a semantics is that a null value can be interpreted as an unknown, inapplicable or even withheld value. Different null constants can be used for each of these different interpretations [28]. Also the use of more than one null value (of the same kind) has been suggested [33], but in this case every new null value uses a new fresh constant; for them the unique names assumption does not apply. The latter alternative allows to keep a relationship between null values in different attributes or relations. However commercial DBMSs consider only one null value, represented by a single constant, that can be given any of the interpretations mentioned above.

\[^3\text{Note that for every } c \in C(G(IC_U)) \text{ it holds } c \in C(G(IC))\]
In [9] a semantics for null values was adopted, according to which a tuple with a null value in any of its attributes would not be the cause for any inconsistencies. In other words, it would not be necessary to check tuples with null values wrt possible violations of ICs (except for “NOT NULL constraints”, of course). This assumption is consistent in some cases with the practice of DBMSs, e.g. in IBM DB2 UDB. Here we will propose a semantics that is less liberal in relation to the participation of null values in inconsistencies; a sort of compromise solution considering the different alternatives available.

Example 4. For IC containing only $ic_1: P(x, y, z) \rightarrow R(y, z)$, the database $D = \{P(a, b, \text{null})\}$ would be: (a) consistent wrt the semantics in [9] because there is a null value in the tuple; (b) consistent wrt the simple semantics of SQL:2003 [23], because there is a null value in one of the set of attributes that are relevant to check the constraint, namely $\{Y, Z\}$; (c) inconsistent wrt the two other semantics in SQL:2003, because there is no tuple in $R$ with a value $b$ in its first attribute.

If we consider, instead of $ic_1$, the constraint $ic_2: P(x, y, z) \rightarrow R(x, y)$, the same database would be consistent only for the semantics in [9], because the other semantics consider only the null value in the attributes that are relevant to check the constraint, and in this case there is no null value there.

We want a null-value semantics that generalizes the semantics defined in SQL:2003 [23] and used by DBMSs, like IBM DB2 UDB. For this reason we consider only one kind of null value, that is interpreted in the same way for different types of ICs, which makes our semantics uniform for a wide class of ICs.

This semantics of IC satisfaction with null values allows us to integrate our results in a compatible way with current commercial implementations; in the sense that the database repairs we will introduce later on would be accepted as consistent by current commercial implementations (for the classes of constraints that can be defined and maintained by them). Because of space limitations we will restrict our presentation of the null values semantics to what is necessary for the rest of this work. A full account and details of the notion of IC satisfaction in relational databases with null values will be left for an extended version.

We need to introduce first a notion of relevance of attributes wrt the occurrence of null values in constraints. Roughly speaking, a constraint is satisfied if any of the relevant attributes has a null value or the constraint is satisfied in the traditional way (no null values involved).

Definition 2. If in the sentence $\psi$ in (1) the variables $x_i, y_j, \ldots$ are replaced by the corresponding attributes, say $X_i, Y_j, \ldots$, specified in the schema $\Sigma$, then the set $A$ of relevant attributes for the integrity constraint (1) is given by:

$$A(\psi) = \{X \mid X \in X_i \text{ and } X \in X_j, \text{ for } P_i \text{ and } P_j \text{ (or } Q_i \text{ and } Q_j \text{) in (1) and } i \neq j\} \cup \bigcup_{j=1}^{n} Y_j \cup \{X \mid X \text{ appears in } \varphi\}.\quad \Box$$

That is, the relevant attributes for a constraint are those involved in joins, those appearing both in the antecedent and consequent of (1), and those in $\varphi$.

Definition 3. For a set of attributes $T$ and a predicate $P \in R$, we denote by $P^T$ the predicate $P$ restricted to the attributes in $T$. $D^T$ denotes the database $D$ with all its database atoms projected onto the attributes in $T$, i.e.

$$D^T = \{P^T(\Pi_T(i)) \mid P(i) \in D\},$$

where $\Pi_T(i)$ is the projection on $T$ of tuple $i$.\quad \Box

4 To simplify the presentation, whenever we have an IC written as a first-order sentence $\varphi(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the variables used in it, we will denote with $X_1, \ldots, X_n$ the corresponding attributes of the relational schema.
Example 5. Consider a universal IC $\forall xyz (P(x, y, z) \rightarrow R(x, y))$. In this case, $A = \{X, Y\}$. We wonder if the value in $Z$ is relevant to check the satisfaction of the constraint in a database $D$. It should not, because we only want to make sure that the values in the first two attributes in $P$ also appear in $R$. Then, checking this is equivalent to checking if $\forall xy (P^A(x, y) \rightarrow R^A(x, y))$ is satisfied by $D^A$.

For a more complex constraint, such as $\forall xyz (P(x, y, z) \wedge S(z, v, w) \wedge v > 3 \rightarrow R(x, y))$, attributes $X$ and $Y$ are relevant to check the implication. $Z$ is needed to do the join, and $V$ is needed to check the comparison. In this case, $A = \{X, Y, Z, V\}$. □

We introduce a special predicate $IsNull(\cdot)$, with $IsNull(c)$ true iff $c$ is null, instead of using the built-in comparison atom $c = null$, because in traditional DBMS this equality would be always evaluated as unknown (as observed in [32], the unique names assumption does not apply to null values).

Definition 4. The constraint $\psi$ in (1) is satisfied in the database instance $D$, denoted $D \models \psi$, if the sentence

$$\forall \bar{x} \left( \bigvee_{x_i \in A(\psi)} IsNull(x_i) \vee \left( \bigwedge_{i=1}^m P_i^{A(\psi)}(\bar{x}_i) \rightarrow \exists \bar{z}_j \bigwedge_{j=1}^n Q_j^{A(\psi)}(\bar{y}_j, \bar{z}_j) \vee \varphi) \right) \right),$$

with $\bar{x} = \bigcup_{i=1}^m \bar{x}_i$, is satisfied in $D^{A(\psi)}$ as usual in first-order logic and treating null as any other constant in $U$, the domain of $D^{A(\psi)}$ (and $D$).

We can see from Definition 4 that we have basically two cases for constraint satisfaction: (a) If there is a null in any of the relevant attributes, then the constraint is satisfied. (b) If no null values appear in them, then the second part of formula (4) has to be checked, i.e., the IC restricted to the relevant attributes. Since there is no null value in them, the constraint can be checked as usual.

As mentioned before, the semantics for IC satisfaction introduced in [9] considered that tuples with null never generated any inconsistencies, even when the null value was not in a relevant attribute. For example, under the semantics in [9], the instance $\{P(b, null)\}$ would be consistent wrt the IC $\forall xy (P(x, y) \rightarrow R(x))$, but it is intuitively clear that there should be a tuple $R(b)$. The new semantics corrects this, and adjusts to the semantics implemented in commercial DBMSs.

Our semantics is a natural extension of the semantics used in commercial DBMSs. Note that: (a) In a DBMS there will never be a join between a null and another value (null or not), (b) Any check constraint with comparison, e.g. $<, >, =$, will never create an inconsistency when comparing a null value with any other value. These two features justify our decision in Definition 2 to include the attributes in the joins and the elements in $\varphi$ among the relevant attributes of the formula, because if there is a null in them an inconsistency will never arise.

Notice that in a database without null values, Definition 4 (so as the definition in [9]) coincides with the traditional, first-order definition of IC satisfaction.

Example 6. Given the ICs: (a) $\forall xyz (P(x, y, z) \rightarrow R(x, y))$, (b) $\forall x (T(x) \rightarrow \exists y z P(x, y, z))$, the database instance $D = \{P(a, d, c), R(a, d), T(a), T(b), P(b, null, g)\}$ is consistent. The set of relevant attributes for (a) is $A_1 = \{X, Y\}$, and for (b) is $A_2 = \{X\}$. When checking the satisfaction of (a), for example with $x = a$ and $y = d$, none of them is a null value, and therefore, we need to check if the sentence $P^{A_1}(a, d) \rightarrow R^{A_1}(a, d)$ is satisfied by $D^{A_1}$. It is satisfied and therefore, for this values of $X$ and $Y$, no inconsistencies are generated. For $x = b$ and $y = null$ (a) is satisfied, because one of the relevant attributes is a null. The same analysis can be done for (b).
If we add to the database the tuple \( P(f, d, \text{null}) \), it would become inconsistent, because for the relevant attributes \( A_1 = \{X, Y\} \), with the values \( f, d \), resp., that are not null values, we need to check if \( (P^{A_1}(f, d) \rightarrow R^{A_1}(f, d)) \) is true in \( (D \cup \{ P(f, d, \text{null}) \})^{A_1} \). Since \( R(f, d) \notin D \cup \{ P(f, d, \text{null}) \} \), the sentence is not satisfied; so \( D \cup \{ P(f, d, \text{null}) \} \) becomes inconsistent.

The predicate \( \text{IsNull} \) also allows us to specify \( \text{NOT NULL} \) constraints (NNCs), which are common in commercial DBMS, and prevent certain attributes from taking a null value. As discussed before, this constraint is different from having \( x \neq \text{null} \). According to our semantics, a NNC can be written as a denial constraint in the form:

\[
\forall \bar{x} (P(\bar{x}) \land \text{IsNull}(x_i) \rightarrow \text{false}),
\]

where \( x_i \in \bar{x} \) is the attribute that cannot take the value null value.

4 Repairs of Incomplete Databases

Let us assume that we have a database instance \( D \), possibly with null values, that is inconsistent, i.e. \( D \) does not satisfy a given set \( IC \) of ICs as defined in Section 3. A repair of \( D \) will be a new instance with the same schema as \( D \) that satisfies \( IC \) and minimally differs from \( D \).

More formally, for database instances \( D, D' \) over the same schema, the distance between them was defined in [2] by \( \Delta(D, D') = (D \setminus D') \cup (D' \setminus D) \). Correspondingly, a repair of \( D \) wrt \( IC \) was defined as an instance \( D' \) that minimizes \( \Delta(D, D') \) under set inclusion and satisfies \( IC \). Finally, a tuple \( t \) was defined as a consistent answer to a query \( Q(\bar{x}) \) in \( D \) wrt \( IC \) if \( t \) is an answer to \( Q(\bar{x}) \) from every of \( D \) wrt \( IC \). The definition of repair given in [2] implicitly ignored the possible presence of null values in that framework. Similarly, in [3, 5, 10], that followed the repair semantics in [2], no null values were used in repairs.

Example 7. Consider the database \( D \) below and the RIC: \( \text{Course}(ID, \text{Code}) \rightarrow \exists \text{Name} \text{Student}(ID, \text{Name}) \). \( D \) is inconsistent, because there is no tuple in \( \text{Student} \) for tuple \( \text{Course}(34, C18) \) in \( \text{Course} \). The database can be minimally repaired by deleting the inconsistent tuple or by inserting a new tuple into table \( \text{Student} \). In the latter case, since the value for attribute \( \text{Name} \) is unknown, we should consider repairs with all the possible values in the domain. Therefore, for the repair semantics introduced in [2], the repairs are of the following two forms:

\[
\begin{array}{c|c} 
\text{Course} & \text{Student} \\
ID & ID & \text{Name} \\
\hline 
27 & C15 & 27 \text{ Ann} \\
34 & C18 & 45 \text{ Paul} \\
\end{array}
\]

for all the possible values of \( \mu \) in the domain, obtaining a possibly infinite number of repairs.

The problem of deciding if a tuple is a consistent answer to a query wrt to a set of universal and referential ICs is undecidable for this repair semantics [10].

An alternative approach is to consider that, in a way, the value \( \mu \) in Example 7 is an unknown value, and therefore, instead of making it take all the values in the domain, we could use it as a null value. We will pursue this idea, which requires to modify the notion of repair accordingly. It will turn out that consistent query answering will become decidable for universal and referential constraints.
Example 8. (example 7 continued) By using null values, there will be only two repairs:

<table>
<thead>
<tr>
<th>Course</th>
<th>ID</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21</td>
<td>C15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21</td>
<td>Ann</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>ID</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>34</td>
<td>C18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>34</td>
<td>null</td>
</tr>
</tbody>
</table>

Here \( \text{null} \) tells us that there is a tuple with 34 in the first attribute, but unknown value in the second.

Now we define in precise terms the notion of repair of a database with null values.

**Definition 5.** [6] Let \( D, D', D'' \) be database instances over the same schema and domain \( U \). It holds \( D' \preceq_D D'' \) iff: (a) For every database atom \( P(\bar{a}) \in \Delta(D, D') \), with \( \bar{a} \in (U \smallsetminus \{\text{null}\}) \),\(^5\) it holds \( P(\bar{a}) \in \Delta(D, D'') \); and (b) For every atom \( Q(\bar{a}, \text{null}) \in \Delta(D, D') \), with \( \bar{a} \in (U \smallsetminus \{\text{null}\}) \), it holds \( Q(\bar{a}, \bar{b}) \in \Delta(D, D'') \) with \( \bar{b} \in U \).

**Definition 6.** Given a database instance \( D \) and a set of universal and referential integrity constraints \( IC \), a repair of \( D \) wrt \( IC \) is a database instance \( D' \) over the same schema, such that \( D' \) satisfies \( IC \) and \( D' \) is \( \preceq_D \)-minimal in the class of database instances that satisfy \( IC \) and share the schema with \( D \). The set of repairs of \( D \) wrt \( IC \) will be denoted with \( \text{Rep}(D, IC) \).

In the absence of null values, this definition of repair coincides with the one given in [2]. It is a consequence of the definition that repairs of violations of UICs are obtained by either deleting or adding database atoms without introducing any null values; and violations of RICs are obtained by either deleting the atom in the antecedent that is generating the inconsistency or by adding an atom with null values.

**Example 9.** If the database instance is \( \{P(a)\} \) and \( IC \) consists only of \( P(x) \rightarrow \exists y Q(x, y) \), then \( \{P(a), Q(a, \text{null})\} \) will be a repair, but not \( \{P(a), Q(a, b)\} \) for any \( b \in D \) different from \( \text{null} \).

In the repair process, one could introduce new inconsistencies when repairing a violation, but we do not want this new inconsistencies to be generated by the introduction of nulls values, the nulls should not be propagated. For our semantics this will never happen: Null values in relevant attributes are not propagated when repairing, because the ICs are satisfied (unless it is a NOT NULL constraint). Tuples with null values in non-relevant attributes can generate inconsistencies (nulls are treated as any other constant), but since the NULL values are in non-relevant attributes they will not propagate.

**Example 10.** Consider the RIC: \( Course(x, y, z) \rightarrow \exists w Prof(z, w) \) and the database instance \( D \) below. The set of relevant attributes of the RIC is \( A = \{ID\} \). Since tuple \( \text{Course}(C50, W05, \text{null}) \) has \( \text{null} \) in the relevant attribute ID, it cannot create an inconsistency, and so it is not propagated to table \( Prof \) because there is no in-

---

\(^5\) That \( \bar{a} \in (U \smallsetminus \{\text{null}\}) \) means that each of the elements in tuple \( \bar{a} \) belongs to \( (U \smallsetminus \{\text{null}\}) \).

\(^6\) \( \text{null} \) is a tuple of null values, that, to simplify the presentation, are placed in the last attributes of \( Q \), but could be anywhere else in \( Q \).
consistency to restore. On the other hand, tuple Course(C18, null, 34) has null in a non-relevant attribute, therefore we need to check if the value of attribute ID, i.e. 34, is in ID of table Prof. It is not, so we have an inconsistency, but null does not propagate to Prof either. Now, in order to repair we have two alternatives: (a) Delete the tuple from Course, or (b) Add tuple (34, null) to Prof.

**Example 11.** Consider the UIC $\forall xy(P(x, y) \rightarrow R(x))$, the RIC $\forall x(T(x) \rightarrow \exists y P(x, y))$, and the database $D = \{P(a, b), P(\text{null}, a), T(c)\}$. The repairs are

<table>
<thead>
<tr>
<th>$i$</th>
<th>$D_i$</th>
<th>$\Delta(D, D_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${P(a, b), P(\text{null}, a), R(a), T(c), P(c, \text{null}), R(c)}$</td>
<td>${R(a), P(c, \text{null}), R(c)}$</td>
</tr>
<tr>
<td>2</td>
<td>${P(a, b), P(\text{null}, a), R(a)}$</td>
<td>${T(c), R(a)}$</td>
</tr>
<tr>
<td>3</td>
<td>${P(\text{null}, a), T(c), P(c, \text{null}), R(c)}$</td>
<td>${P(a, b), P(c, \text{null}), R(c)}$</td>
</tr>
<tr>
<td>4</td>
<td>${P(\text{null}, a)}$</td>
<td>${P(a, b), T(c)}$</td>
</tr>
</tbody>
</table>

It can be seen that, in the repairs, null in $P(\text{null}, a)$ does not propagate through the universal constraint to $R(\text{null})$. Notice that the additional instance $D_5 = \{P(\text{null}, a), P(a, b), R(a), T(c), P(c, a), R(c)\}$, with $P(c, a)$ introduced in order to satisfy the RIC, and $R(c)$ to satisfy the UIC, does satisfy IC, but is not a repair because $\Delta(D, D_1) < \Delta(D, D_5) = \{R(a), P(c, a), R(c)\}$. □

**Example 12.** Consider a schema with a table $R(X, Y)$, with primary key $X$, and a table $S(U, V)$, with $V$ a foreign key to table $R$. The constraints can be written in the form (1) as $\forall xyz (R(x, y) \land R(x, z) \rightarrow y = z)$ and $\forall uv (S(u, v) \rightarrow \exists y R(v, y))$. $D = \{R(a, b), R(a, c), S(e, f), S(\text{null}, a)\}$ is inconsistent, and its repairs are

<table>
<thead>
<tr>
<th>$i$</th>
<th>$D_i$</th>
<th>$\Delta(D, D_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${R(a, b), S(e, f), S(\text{null}, a), R(f, \text{null})}$</td>
<td>${R(a, c), R(f, \text{null})}$</td>
</tr>
<tr>
<td>2</td>
<td>${R(a, c), S(e, f), S(\text{null}, a), R(f, \text{null})}$</td>
<td>${R(a, b), R(f, \text{null})}$</td>
</tr>
<tr>
<td>3</td>
<td>${R(a, b), S(\text{null}, a)}$</td>
<td>${R(a, c), S(e, f)}$</td>
</tr>
<tr>
<td>4</td>
<td>${R(a, c), S(\text{null}, a)}$</td>
<td>${R(a, b), S(e, f)}$</td>
</tr>
</tbody>
</table>

If a given database $D$ is consistent wrt a set of ICs, then there is only one repair, that coincides with $D$.

The following example shows what can happen if we use a NNC in combination with a referential constraint.

**Example 13.** Consider the database $D = \{P(a), P(b), Q(b, c)\}$, the RIC $\forall x (P(x) \rightarrow \exists y Q(x, y))$, and the NNC $\forall xy(Q(x, y) \land IsNull(y) \rightarrow false)$. In this case, we have a NNC on an existentially quantified attribute in the RIC. We cannot repair as expected, i.e. using null values. Actually, the repairs are $\{P(b), Q(b, c)\}$, corresponding to a tuple deletion, but also those of the form $\{P(a), P(b), Q(b, c), Q(a, \mu)\}$, for every $\mu \in U \setminus \{\text{null}\}$, that are obtained by multiple tuple insertions. Thus, we essentially recover the repair semantics in [2]. □

The repair semantics above could be modified in order to repair only through tuple deletions, avoiding having to repair through multiple tuple insertions, when null values cannot be used due to the presence of conflicting NNCs. This could be done as follows:

<table>
<thead>
<tr>
<th>Course</th>
<th>Code</th>
<th>Term</th>
<th>ID</th>
<th>Prof</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>C27</td>
<td>W04</td>
<td>21</td>
<td>null</td>
<td>Ann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C50</td>
<td>W05</td>
<td>null</td>
<td>null</td>
<td>Paul</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As we can see, null values do not propagate in the repair process. □

<table>
<thead>
<tr>
<th>Course</th>
<th>Code</th>
<th>Term</th>
<th>ID</th>
<th>Prof</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>C27</td>
<td>W04</td>
<td>21</td>
<td>null</td>
<td>Ann</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C50</td>
<td>W05</td>
<td>34</td>
<td>null</td>
<td>Paul</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If $Rep(D, IC)$ is the class of repairs according to Definitions 5 and 6, the alternative class of repairs, $Rep_d(D, IC)$, that prefers tuple deletions over multiple insertions, can be defined by $Rep_d(D, IC) := \{D' \mid D' \in Rep(D, IC) \text{ and there is no } D'' \in Rep(D, IC') \text{ with } D'' <_D D'\}$, where $IC'$ is $IC$ without the (conflicting) NNCs. Actually, the repair programs introduced in Section 5 compute wrt the $Rep_d$-semantics. Since the first semantics is easier to deal with, and motivated by Example 13, in the following: We will assume that our sets of ICs never have a NNC on an attribute that is existentially quantified in a RIC. In this way, we will always be able to repair RICs using null values or deletions. If there are no (conflicting) NNCs, the two semantics coincide.

Notice that with our repair semantics, we can always restrict the database domain $U$ to $\text{adom}(D) \cup \{\text{null}\}$, where $\text{adom}(D)$ is the active domain of the original instance $D$. Since we are assuming that the set of integrity constraints $IC$ is consistent, there always exists a repair of a database $D$ wrt $IC$. Also, for the semantics introduced here, the set of repairs is finite and each of them is finite (i.e. every database relation in it has a finite extension).

**Proposition 1.** Given a database instance $D$ and a finite and logically consistent set $IC$ consisting of RICs, UICs and NNCs, the set $Rep(D, IC)$ is non-empty and finite; and every $D' \in Rep(D, IC)$ is finite.7

**Definition 7.** [2] Given a database $D$, a set of ICs $IC$, and a query $Q(\bar{x})$, a ground tuple $t$ is a consistent answer to $Q$ wrt $IC$ in $D$ iff for every $D' \in Rep(D, IC)$, $D' \models Q[t]$. If $Q$ is a sentence (boolean query), then yes is a consistent answer iff $D' \models Q$ for every $D' \in Rep(D, IC)$. Otherwise, the consistent answer is no.

**Example 14.** Given the IC $\forall x(T(x) \rightarrow \exists y P(x, y))$ and the inconsistent database $D = \{P(a, d), R(a, d), T(a), T(b), R(b, e)\}$, we have $Rep(D, IC) = \{\{P(a, d), R(a, d), T(a), R(b, e)\}, \{P(a, d), R(a, d), T(a), T(b), R(b, e), P(b, \text{null})\}\}$. For the queries $Q_1: P(x, y)$ and $Q_2: \forall x, y P(x, y) \rightarrow \exists z R(x, z)$, the consistent answer to $Q_1$ is $(a, d)$ (it is the only tuple in $P$ in both repairs). For $Q_2$ the consistent answer is yes, because the sentence is true in both repairs.

## 5 Repair Logic Programs

The stable models semantics was introduced in [19] to give a semantics to disjunctive logic programs that are non-stratified, i.e. that contain recursive definitions that contain weak negation. By now it is the standard semantics for such programs. Under this semantics, a program may have several stable models; and what is true of the program is what is true in all its stable models (a cautious semantics).

Repairs of relational databases can be specified as stable models of disjunctive logic programs. In [6, 9, 11] such programs were presented, but they were based on classic IC satisfaction, that differs from the one introduced in Section 3. A limitation of those repairs programs is that they are sound and complete for repairs of RIC-acyclic constraints. Otherwise, we can only ensure completeness (every repair is a stable model), but we may have stable models for which the associated database is consistent, but does not minimally differ from the original database.

The repair programs we will present now not only implement the repair semantics introduced in Section 3, but there is a one-to-one correspondence between their stable models and the repairs they specify, even for cyclic sets of constraints. The repair

7 For proofs of all results go to www.scs.carleton.ca/~lbravo/RICdemos.pdf
programs use annotation constants with the intended, informal semantics shown in the table below (the exact semantics is captured by the stable model semantics of the programs that use them). The annotations are used in an extra attribute introduced in each database predicate; so for a predicate \( P \in R \), the version of it that is expanded with the extra attribute is denoted by \( P^a \).

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Atom</th>
<th>The tuple ( P^a ) is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_a )</td>
<td>( P^a(t_a, t_a) )</td>
<td>advised to be made true</td>
</tr>
<tr>
<td>( f_a )</td>
<td>( P^a(f_a, t_a) )</td>
<td>advised to be made false</td>
</tr>
<tr>
<td>( t^* )</td>
<td>( P^a(t^<em>, t^</em>) )</td>
<td>true or becomes true</td>
</tr>
<tr>
<td>( t^{**} )</td>
<td>( P^a(t^{<strong>}, t^{</strong>}) )</td>
<td>it is true in the repair</td>
</tr>
</tbody>
</table>

In the repair program, \( null \) is treated as any other constant in \( U \), and therefore the \( IsNull(x) \) atom can be replaced by \( x = null \).

**Definition 8.** Given a database instance \( D \), a set \( IC \) of UICs, RICs and NNCs, the repair program \( \Pi(D, IC) \) contains the following rules:

1. Facts: \( P^a(\bar{a}) \) for each atom \( P(\bar{a}) \in D \).
2. For every UIC \( \psi \) of form (2), the rules:
   \[
   \bigwedge_{i=1}^m P^a(\bar{x}_i, \bar{a}_i) \vee \bigwedge_{j=1}^n Q^a(\bar{y}_j, \bar{a}_a) \leftarrow \bigwedge_{i=1}^m P^a(\bar{x}_i, t^*); \bigwedge_{Q^a \in Q'} Q^a(\bar{y}_j), \bigwedge_{A \in \text{Aux}^\psi} \text{not } Q^a(\bar{y}_j); \bigwedge_{X \in \text{Aux}^\psi} x_t \neq \text{null}, \forall \varphi,
   \]
   for every set \( Q' \) and set \( Q'' \) such that \( Q' \cup Q'' = \bigcup_{i=1}^m Q_i \) and \( Q' \cap Q'' = \emptyset \). Here \( \text{Aux}^\psi \) is the set of relevant attributes for \( \psi \), and \( \varphi \) is a conjunction of built-ins that is equivalent to the negation of \( \varphi \).
3. For every RIC of form (3), the rules:
   \[
   P^a(\bar{x}_i, \bar{a}_a) \vee Q(\bar{x}', null, \bar{t}_a) \leftarrow P^a(\bar{x}, t^*), \text{not aux}_1(\bar{x}'), \text{not aux}_2(\bar{x}'), \bar{x}' \neq \text{null},
   \]
   \[
   aux_1(\bar{x}') \leftarrow P^a(\bar{x}, t^{**}), Q(\bar{x}', x', \bar{t}^{**}), \bar{x} \neq \text{null}, \bar{y} \neq \text{null},
   \]
   \[
   aux_2(\bar{x}') \leftarrow Q(\bar{x}', \bar{y}', \bar{t}^{**}), \bar{x}' \neq \text{null}, \bar{y} \neq \text{null}.
   \]
   for every \( y_i \in \bar{y} \).
4. For every NNC of form (5), the rule:
   \[
   P^a(\bar{x}_i, \bar{a}_a) \leftarrow P^a(\bar{x}, t^*), x_t = \text{null}.
   \]
5. For each predicate \( P \in R \), the annotation rules:
   \[
   P^a(\bar{x}), t^*) \leftarrow P^a(\bar{x}), \bar{t}_a.
   \]
6. For every predicate \( P \in R \), the interpretation rule:
   \[
   P^a(\bar{x}, \bar{t}^{**}) \leftarrow P^a(\bar{x}, t^*), \text{not } P^a(\bar{x}, \bar{t}_a).
   \]
7. For every predicate \( P \in R \), the program denial constraint:
   \[
   \leftarrow P^a(\bar{x}, \bar{t}_a), P^a(\bar{x}, \bar{t}_a).
   \]

Facts in 1. are the elements of the database. Rules 2., 3. and 4. capture, in the right-hand side, the violation of ICs of the forms (2), (3), and 5. resp., and, with the left-hand side, the intended way of restoring consistency. The auxiliary predicates \( Q' \) and \( Q'' \) are used to check that in all the possible combinations, the consequent of a UIC is not being satisfied. Since the satisfaction of UICs and RICs needs to be checked only if none of the relevant attributes is \( null \), we use \( x \neq null \) in rule 2 and in the first two rules in 3. (as usual, \( x' \neq null \) means the conjunction of the atoms \( x_j \neq null \) for \( x_j \in \bar{x}' \)). Notice that rules 3. are implicitly based on the fact that the relevant attributes for a RIC of the form (3) are \( A = \{ X \mid x \in \bar{x}' \} \). Rules 5. capture the atoms that are part of the inconsistent database or that become true in the repair process; and rules 6. those that become true in the repairs. Rule 7. enforces, by discarding models, that no atom can be made both true and false in a repair.

**Example 15.** (example 11 continued) The repair program \( \Pi(D, IC) \) is the following:

1. \( P(a, b), P(null, a), T(c) \).
2. \( P(x, y, \bar{f}_a) \vee R(x, \bar{t}_a) \leftarrow P(x, y, t^*), R(x, \bar{f}_a), x \neq null \).
   \[
   P(x, y, \bar{f}_a) \vee R(x, \bar{t}_a) \leftarrow P(x, y, t^*), \text{not } R(x), x \neq null.
   \]
3. \( T(x, f_a) \lor P(x, null, t_a) \leftarrow T(x, t^\ast), \text{not } aux_1(x), \text{ not } aux_2(x), x \neq null. \)
   \( aux_1(x) \leftarrow T(x, t^\ast), P(x, y, t^\ast), x \neq null, y \neq null. \)
   \( aux_2(x) \leftarrow P(x, y, t^\ast), x \neq null, y \neq null. \)
4. \( P(x, y, t^\ast) \leftarrow P(x, y, t_a). \quad P(x, y, t^\ast) \leftarrow P(x, y, t_a). \) (similarly for \( R \) and \( T \))
5. \( P(x, y, t^\ast) \leftarrow P(x, y, t_a). \quad P(x, y, t^\ast) \leftarrow P(x, y, t_a). \) (similarly for \( R \) and \( T \))
6. \( P(x, y, t^\ast) \leftarrow P(x, y, t_a). \quad P(x, y, t^\ast) \leftarrow P(x, y, t_a). \) (similarly for \( R \) and \( T \))
7. \( \leftarrow P(x, t_a), P(x, f_a). \quad \leftarrow R(x, t_a), R(x, f_a). \quad \leftarrow T(x, t_a), T(x, f_a). \)

Only rules 2. and 3. depend on the ICs: rules 2. for the UIC, and 3. for the RIC. They say how to repair the inconsistencies. Note that for RICs in which \( \bar{y} \) consists only on one variable, there is no need for \( aux_2 \) since \( aux_1 \) will create the same effect over the stable models. There is no rule 4. because there is no NNC.

The repairs can be obtained by collecting the atoms annotated with \( t^\ast \) in the stable models of the program.

**Definition 9.** Let \( \mathcal{M} \) be a stable model of program \( \Pi(D, IC) \). The database instance associated with \( \mathcal{M} \) is \( D_\mathcal{M} = \{ P(a) \mid P \in \mathcal{R} \text{ and } P(a, t^\ast) \in \mathcal{M} \} \).

**Example 16.** (example 15 continued) The program has four stable models (the facts of the program are omitted for simplicity):

\[
\mathcal{M}_1 = \{ P(a, b, t^\ast), P(null, a, t^\ast), T(c, t^\ast), aux_2(a), P(a, b, t^\ast), P(null, a, t^\ast), T(c, t^\ast), aux_2(a), R(a, t_a), R(c, t^\ast), \}
\]

\[
\mathcal{M}_2 = \{ P(a, b, t^\ast), P(null, a, t^\ast), T(c, t^\ast), aux_2(a), R(a, t_a), P(c, null, t_a), P(c, null, t^\ast), P(c, null, t^\ast), \}
\]

\[
\mathcal{M}_3 = \{ P(a, b, t^\ast), P(null, a, t^\ast), T(c, t^\ast), P(c, null, t_a), P(c, null, t^\ast), \}
\]

\[
\mathcal{M}_4 = \{ P(a, b, t^\ast), P(null, a, t^\ast), T(c, t^\ast), P(a, b, f_a), P(null, a, t^\ast), T(c, f_a) \}. \]

The databases associated to the models select the underlined atoms: \( D_1 = \{ P(a, b), P(null, a), R(a) \}, D_2 = \{ P(a, b), P(null, a), T(c), P(c, null), R(a), R(c) \}, D_3 = \{ P(null, a), T(c), P(c, null), R(c) \} \) and \( D_4 = \{ P(null, a) \} \). As expected these are the repairs obtained in Example 11.

**Example 17.** (example 13 continued) The associated repair program is:

1. \( P(a) \quad P(b) \quad Q(b, c) \)
2. \( P(x, t^\ast) \leftarrow P(x, t_a). \quad P(x, t^\ast) \leftarrow P(x). \)
3. \( Q(x, y, t^\ast) \leftarrow Q(x, y, t_a). \quad Q(x, y, t^\ast) \leftarrow Q(x, y). \)
4. \( P(x, f_a) \lor Q(x, null, t_a) \leftarrow P(x, t^\ast), \text{not } aux_1(x), \text{ not } aux_2(x), x \neq null. \)
   \( aux_1(x) \leftarrow P(x, t^\ast), Q(x, y, t^\ast), x \neq null, y \neq null. \)
   \( aux_2(x) \leftarrow Q(x, y, t^\ast), x \neq null, y \neq null. \)
5. \( Q(x, y, f_a) \leftarrow Q(x, y, t^\ast), y = null. \)
6. \( P(x, t^\ast) \leftarrow P(x, t_a). \quad P(x, t^\ast) \leftarrow P(x). \quad \text{not } P(x, f_a). \)
   \( Q(x, y, t^\ast) \leftarrow Q(x, y, t_a). \quad Q(x, y, t^\ast) \leftarrow Q(x, y, f_a). \quad \text{not } Q(x, y, f_a). \)
7. \( \leftarrow P(x, t_a), P(x, f_a). \quad \leftarrow Q(x, y, t_a), Q(x, y, f_a). \)

There is a single stable model: \( \mathcal{M} = \{ P(a, t^\ast), P(b, t^\ast), Q(b, c, t^\ast), aux_1(b), aux_2(b), P(a, f_a), P(b, t^\ast), Q(b, c, t^\ast) \} \), with \( D_\mathcal{M} = \{ P(b), Q(b, c) \} \) (the only repair).

**Theorem 1.** Let \( IC \) be a set of UICs, RICs and NNCs. If \( \mathcal{M} \) is a stable model of \( \Pi(D, IC) \), then \( D_\mathcal{M} \) is a repair of \( D \) with respect to \( IC \). Furthermore, the repairs obtained in this way are all the repairs of \( D \).
In order to obtain the consistent answers to a query $Q$, the latter has to be expressed as a query program. If $Q$ is first-order, this is a standard process [31, 1]. The query program is run together with the program that specifies the repairs. This evaluation can be implemented on top of systems like DLV, a logic programming system that computes the stable models semantics [26].

**Example 18.** (example 11, 15 and 16 continued) To compute the consistent answers to the query $Q(y): \exists x P(x, y)$, we first express the query as a logic program: $Ans(y) \leftarrow P(x, y, t^*)$, that can be added to the repair program, whose stable models become expanded with $Ans$ atoms, obtaining: $M_1' = M_1 \cup \{Ans(b), Ans(a)\}$; $M_2' = M_2 \cup \{Ans(b), Ans(a), Ans(null)\}$; $M_3' = M_3 \cup \{Ans(a), Ans(null)\}$; $M_4' = M_4 \cup \{Ans(a)\}$. Since the consistent answers are the ones we get from all the possible repairs, the only consistent answer to the query is $\{a\}$. □

Repairs of a database always exist and they can be exactly captured by the repair programs for the ICs we have considered. In consequence, stable models of the repair program always exist, are finite in number, and all finite as models (cf. Proposition 1). Furthermore, the repair program combined with the query program is a split program [30]. In this case, the problem of computing the cautious answers under the stable semantics for a Datalog$^\sim \lor$ query is decidable [17], in contrast to what happens with the classic repair semantics [2], as established in [10].

**Theorem 2.** Consistent query answering for Datalog$^\sim \lor$ queries wrt to finite sets of UICs, RICs and NNCs under the repair semantics introduced in Section 4 is decidable. □

### 6 Head-Cycle-Free Programs

In some cases, the repair programs introduced in Section 5 can be transformed into equivalent non-disjunctive programs. This is the case when they become head-cycle-free [7]. Query evaluation from such programs has lower computational complexity than general disjunctive programs, actually the data complexity is reduced from $\Pi^P_2$-complete to coNP-complete [7, 16, 25]. We briefly recall their definition.

The dependency graph of a ground disjunctive program $\Pi$ is the directed graph that has ground atoms as vertices, and an edge from atom $A$ to atom $B$ if there is a rule with $A$ (positive) in the body and $B$ (positive) in the head. $\Pi$ is head-cycle free (HCF) if its dependency graph does not contain any directed cycles passing through two atoms in the head of the same rule. A disjunctive program $\Pi$ is HCF if its ground version is HCF.

A HCF program $\Pi$ can be transformed into a non-disjunctive normal program $sh(\Pi)$ that has the same stable models. It is obtained by replacing every disjunctive rule of the form $\bigvee_{i=1}^m P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^n Q_j(\bar{y}_j), \varphi$, by the $n$ rules $P_i(\bar{x}_i) \leftarrow \bigwedge_{j=1}^n Q_j(\bar{y}_j), \varphi, \bigwedge_{k \neq i} \neg P_k(\bar{x}_k)$, for $i = 1, \ldots, n$.

For certain classes of queries and ICs, consistent query answering has a data complexity lower than $\Pi^P_2$ (a sharp lower bound in the general case) [13]. In those cases, it is natural to consider this kind of transformations of the disjunctive repair program. In the rest of this section we will consider sets $IC$ of integrity constraints formed by UICs, RICs and NNCs.

**Definition 10.** A predicate $P$ is bilateral with respect to $IC$ if it belongs to the antecedent of a constraint $ic_1 \in IC$ and to the consequent of a constraint $ic_2 \in IC$, where $ic_1$ and $ic_2$ are not necessarily different. □
Example 19. If $IC = \{ \forall x \ (T(x) \rightarrow \exists y \ R(x,y)) , \forall x y \ (S(x,y) \rightarrow T(x)) \}$, the only bilateral predicate is $T$.

Theorem 3. For a set $IC$ of UICs, RICs and NNCs, if for every $ic \in IC$, it holds that (a) $ic$ has no bilateral predicates; or (b) $ic$ has exactly one occurrence of a bilateral predicate (without repetitions), then the program $\Pi(D, IC)$ is HCF.

For example, if in $IC$ we have the constraint $P(x,y) \rightarrow P(y,x)$, then $P$ is a bilateral predicate, and the condition in the theorem is not satisfied. Actually, the program $\Pi(D, IC)$ is not HCF. If we have instead $P(x,a) \rightarrow P(x,b)$, even though the condition is not satisfied, the program is HCF. Therefore, the condition is sufficient, but not necessary for the program to be HCF.

This theorem can be immediately applied to useful classes of ICs, like denial constraints, because they do not have any bilateral literals, and in consequence, the repair program is HCF.

Corollary 1. If $IC$ contains only constraints of the form $\bigwedge_{i=1}^n P_i(t_i) \rightarrow \varphi$, where $P_i(t_i)$ is a database atom and $\varphi$ is a formula containing built-in predicates only, then $\Pi(D, IC)$ is HCF.

This result is consistent with the intrinsic complexity of CQA for functional dependencies, to which Corollary 1 applies: The data complexity becomes polynomial or $coNP$-complete [15].

Acknowledgments: Research supported by NSERC, CITO/IBM-CAS Student Internship Program. L. Bertossi is Faculty Fellow of IBM Center for Advanced Studies (Toronto Lab.).

References


Models for Incomplete and Probabilistic Information

Todd J. Green and Val Tannen
University of Pennsylvania
{tjgreen,val}@cis.upenn.edu

Abstract. We discuss, compare and relate some old and some new models for incomplete and probabilistic databases. We characterize the expressive power of \(c\)-tables over infinite domains and we introduce a new kind of result, algebraic completion, for studying less expressive models. By viewing probabilistic models as incompleteness models with additional probability information, we define completeness and closure under query languages of general probabilistic database models and we introduce a new such model, probabilistic \(c\)-tables, that is shown to be complete and closed under the relational algebra.

1 Introduction

The representation of incomplete information in databases has been an important research topic for a long time, see the references in [18], in Ch.19 of [2], in [31], in [35, 25], as well as the recent [33, 30, 29]. Moreover, this work is closely related to recently active research topics such as inconsistent databases and repairs [4], answering queries using views [1], and data exchange [13]. The classic reference on incomplete databases remains [20] with the fundamental concept of \(c\)-table and its restrictions to simpler tables with variables. The most important result of [20] is the query answering algorithm that defines an algebra on \(c\)-tables that corresponds exactly to the usual relational algebra (\(RA\)). A recent paper [29] has defined a hierarchy of incomplete database models based on finite sets of choices and optional inclusion. One of our contributions consists of comparisons between the models [29] and the tables with variables from [20].

Two criteria have been provided for comparisons among all these models: [20, 29] discuss closure under relational algebra operations, while [29] also emphasizes completeness, specifically the ability to represent all finite incomplete databases. We point out that the latter is not appropriate for tables with variables over an infinite domain, and we contribute another criterion, \(RA\)-completeness, that fully characterizes the expressive power of \(c\)-tables.

We also introduce a new idea for the study of models that are not complete. Namely, we consider combining existing models with queries in various fragments of relational algebra. We then ask how big these fragments need to be to obtain a combined model that is complete. We give a number of such algebraic completion results.
Early on, probabilistic models of databases were studied less intensively than incompleteness models, with some notable exceptions [7, 5, 28, 23, 10]. Essential progress was made independently in three papers [15, 22, 34] that were published at about the same time. [15, 34] assume a model in which tuples are taken independently in a relation with given probabilities. [22] assumes a model with a separate distribution for each attribute in each tuple. All three papers attacked the problem of calculating the probability of tuples occurring in query answers. They solved the problem by developing more general models in which rows contain additional information (“event expressions”, “paths”, “traces”), and they noted the similarity with the conditions in $c$-tables.

We go beyond the problem of individual tuples in query answers by defining closure under a query language for probabilistic models. Then we develop a new model, probabilistic $c$-tables that adds to the $c$-tables themselves probability distributions for the values taken by their variables. Here is an example of such a representation that captures the set of instances in which Alice is taking a course that is Math with probability 0.3; Physics (0.3); or Chemistry (0.4), while Bob takes the same course as Alice, provided that course is Physics or Chemistry and Theo takes Math with probability 0.85:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>$x$</td>
<td>$x = \text{phys} \lor x = \text{chem}$</td>
</tr>
<tr>
<td>Bob</td>
<td>$x$</td>
<td>$t = 1$</td>
</tr>
<tr>
<td>Theo</td>
<td>math</td>
<td>$t = 1$</td>
</tr>
</tbody>
</table>

$x = \begin{cases} 
\text{math} : 0.3 \\
\text{phys} : 0.3 \\
\text{chem} : 0.4 \\
\end{cases} 
\quad t = \begin{cases} 
0 : 0.15 \\
1 : 0.85 \\
\end{cases}$

The concept of probabilistic $c$-table allows us to solve the closure problem by using the same algebra on $c$-tables defined in [20].

We also give a completeness result by showing that probabilistic boolean $c$-tables (all variables are two-valued and can appear only in the conditions, not in the tuples) can represent any probabilistic database.

An important conceptual contribution is that we show that, at least for the models we consider, the probabilistic database models can be seen, as probabilistic counterparts of incomplete database models. In an incompleteness model a tuple or an attribute value in a tuple may or may not be in the database. In its probabilistic counterpart, these are seen as elementary events with an assigned probability. For example, the models used in [15, 22, 34] are probabilistic counterparts of the two simplest incompleteness models discussed in [29]. As another example, the model used in [10] can be seen as the probabilistic counterpart of an incompleteness model one in which tuples sharing the same key have an exclusive-or relationship.

A consequence of this observation is that, in particular, query answering for probabilistic $c$-tables will allow us to solve the problem of calculating probabilities about query answers for any model that can be defined as a probabilistic counterpart of the incompleteness models considered in [20, 29].
This paper is purely theoretical. Nonetheless, it was motivated by the work the authors are doing with others on the Orchestra\(^1\) and SHARQ\(^2\) projects. These projects are concerned with certain aspects of collaborative information sharing. Incompleteness arises in Orchestra (a peer-to-peer data exchange system) in the process of update propagation between sites. Incompleteness is also exploited in query answering algorithms. Probabilistic models are used in SHARQ (a bio-informatics data sharing system) to model approximate mappings between schemas used by groups of researchers. The sources of uncertainty here include data from error-prone experiments and accepted scientific hypotheses that allow for the limited mismatch. We expect that the results of this paper will help us in choosing appropriate representation systems that will be used internally in the Orchestra and SHARQ systems.

2 Incomplete Information and Representation Systems

Our starting point is suggested by the work surveyed in [18], in Ch. 19 of [2], and in [31]. A database that provides incomplete information consists of a set of possible instances. At one end of this spectrum we have the conventional single instances, which provide “complete information.” At the other end we have the set of all allowable instances which provides “no information” at all, or “zero information.”

We adopt the formalism of relational databases over a fixed countably infinite domain \(\mathbb{D}\). We use the unnamed form of the relational algebra. To simplify the notation we will work with relational schemas that consist of a single relation name of arity \(n\). Everything we say can be easily reformulated for arbitrary relational schemas. We shall need a notation for the set of all (conventional) instances of this schema, i.e., all the finite \(n\)-ary relations over \(\mathbb{D}\):

\[\mathcal{N} := \{ I \mid I \subseteq \mathbb{D}^n, I \text{ finite} \}\]

**Definition 1.** An incomplete(-information) database (i-database for short), \(\mathcal{I}\), is a set of conventional instances, i.e., a subset \(\mathcal{I} \subseteq \mathcal{N}\).

The usual relational databases correspond to the cases when \(\mathcal{I} = \{I\}\). The no-information or zero-information database consists of all the relations: \(\mathcal{N}\).

Conventional relational instances are finite. However, because \(\mathbb{D}\) is infinite incomplete databases are in general infinite. Hence the interest in finite, syntactical, representations for incomplete information.

**Definition 2.** A representation system consists of a set (usually a syntactically defined “language”) whose elements we call tables, and a function \(\text{Mod}\) that associates to each table \(T\) an incomplete database \(\text{Mod}(T)\).

\(^1\)http://www.cis.upenn.edu/~zives/orchestra

\(^2\)http://db.cis.upenn.edu/projects/SHARQ
The notation corresponds to the fact that \( T \) can be seen as a logical assertion such that the conventional instances in \( \text{Mod}(T) \) are in fact the models of \( T \) (see also \[27, 32\]).

The classical reference \[20\] considers three representation systems: **Codd tables**, \( v \)-tables, and \( c \)-tables. \( v \)-tables are conventional instances in which variables can appear in addition to constants from \( \mathbb{D} \). If \( T \) is a \( v \)-table then

\[
\text{Mod}(T) := \{ \nu(T) \mid \nu : \text{Var}(T) \to \mathbb{D} \text{ is a valuation for the variables of } T \}
\]

Codd tables are \( v \)-tables in which all the variables are distinct. They correspond roughly to the current use of nulls in SQL, while \( v \)-tables model “labeled” or “marked” nulls. \( c \)-tables are \( v \)-tables in which each tuple is associated with a condition — a boolean combination of equalities involving variables and constants. We typically use the letter \( \varphi \) for conditions. The tuple condition is tested for each valuation \( \nu \) and the tuple is discarded from \( \nu(T) \) if the condition is not satisfied.

**Example 1.** Here is an example of a \( v \)-table.

\[
R := \begin{array}{ccc}
1 & 2 & x \\
3 & x & y \\
4 & z & 5 \\
\end{array}
\]

\[
\text{Mod}(R) = \left\{ \begin{array}{ccc}
1 & 2 & 1 \\
3 & 3 & 1 \\
1 & 4 & 5 \\
\end{array}, \begin{array}{ccc}
1 & 2 & 2 \\
3 & 3 & 2 \\
1 & 4 & 5 \\
\end{array}, \ldots, \begin{array}{ccc}
1 & 2 & 77 \\
3 & 77 & 89 \\
97 & 4 & 5 \\
\end{array}, \ldots \right\}
\]

**Example 2.** Here is an example of a \( c \)-table.

\[
S := \begin{array}{ccc}
1 & 2 & x \\
3 & x & y \\
4 & z & 5 \\
\end{array}
\]

\[
x = y \land z \neq 2 \\
z \neq 1 \lor x \neq y \\
\]

\[
\text{Mod}(S) = \left\{ \begin{array}{ccc}
1 & 2 & 1 \\
3 & 1 & 1 \\
1 & 4 & 5 \\
\end{array}, \begin{array}{ccc}
1 & 2 & 2 \\
3 & 1 & 2 \\
1 & 4 & 5 \\
\end{array}, \ldots, \begin{array}{ccc}
1 & 2 & 77 \\
3 & 77 & 89 \\
97 & 4 & 5 \\
\end{array}, \ldots \right\}
\]

Several other representation systems have been proposed in a recent paper \[29\]. We illustrate here three of them and we discuss several others later. A \( ? \)-table is a conventional instance in which tuples are optionally labeled with “?,” meaning that the tuple may be missing. An \( \text{or-set-table} \) looks like a conventional instance but \( \text{or-set} \) values \[21, 26\] are allowed. An \( \text{or-set} \) value \((1, 2, 3)\) signifies that exactly one of 1, 2, or 3 is the “actual” (but unknown) value. Clearly, the two ideas can be combined yielding another representation systems that we might (awkwardly) call \( \text{or-set-?}-\text{tables} \).

**Example 3.** Here is an example of an \( \text{or-set-?}-\text{table} \).

\[
T := \begin{array}{ccc}
1 & 2 & (1, 2) \\
3 & (1, 2) & (3, 4) \\
4, 5 & ? \\
\end{array}
\]

\[
\text{Mod}(T) = \left\{ \begin{array}{ccc}
1 & 2 & 1 \\
3 & 1 & 3 \\
4 & 4 & 5 \\
\end{array}, \begin{array}{ccc}
1 & 2 & 2 \\
3 & 1 & 3 \\
4 & 4 & 5 \\
\end{array}, \ldots, \begin{array}{ccc}
1 & 2 & 2 \\
3 & 1 & 3 \\
4 & 4 & 5 \\
\end{array}, \ldots \right\}
\]
“Completeness” of expressive power is the first obvious question to ask about representation systems. This brings up a fundamental difference between the representation systems of [20] and those of [29]. The presence of variables in a table $T$ and the fact that $\mathbb{D}$ is infinite means that $\text{Mod}(T)$ may be infinite. For the tables considered in [29], $\text{Mod}(T)$ is always finite.

[29] defines completeness as the ability of a representation system to represent “all” possible incomplete databases. For the kind of tables considered in [29] the question makes sense. But in the case of the tables with variables in [20] this is hopeless for trivial reasons. Indeed, in such systems there are only countably many tables while there are uncountably many incomplete databases (the subsets of $\mathbb{N}$, which is infinite). We will discuss separately below finite completeness for systems that only represent finite database. Meanwhile, we will develop a different yardstick for the expressive power of tables with variables that range over an infinite domain.

$c$-tables and their restrictions ($v$-tables and Codd tables) have an inherent limitation: the cardinality of the instances in $\text{Mod}(T)$ is at most the cardinality of $T$. For example, the zero-information database $\mathbb{N}$ cannot be represented with $c$-tables. It also follows that among the incomplete databases that are representable by $c$-tables the “minimal”-information ones are those consisting for some $m$ of all instances of cardinality up to $m$ (which are in fact representable by Codd tables with $m$ rows). Among these, we make special use of the ones of cardinality 1:

$$Z_k := \{\{t\} \mid t \in \mathbb{D}^k\}.$$  

Hence, $Z_k$ consists of all the one-tuple relations of arity $k$. Note that $Z_k = \text{Mod}(Z_k)$ where $Z_k$ is the Codd table consisting of a single row of $k$ distinct variables.

**Definition 3.** An incomplete database $\mathcal{I}$ is $\mathcal{RA}$-definable if there exists a relational algebra query $q$ such that $\mathcal{I} = q(Z_k)$, where $k$ is the arity of the input relation name in $q$.

**Theorem 1.** If $\mathcal{I}$ is an incomplete database representable by a c-table $T$, i.e., $\mathcal{I} = \text{Mod}(T)$, then $\mathcal{I}$ is $\mathcal{RA}$-definable.

**Proof.** Let $T$ be a c-table, and let $\{x_1, \ldots, x_k\}$ denote the variables in $T$. We want to show that there exists a query $q$ in $\mathcal{RA}$ such that $q(\text{Mod}(Z_k)) = \text{Mod}(T)$. Let $n$ be the arity of $T$. For every tuple $t = (a_1, \ldots, a_n)$ in $T$ with condition $\varphi_t$, let $\{x_{i_1}, \ldots, x_{i_j}\}$ be the variables in $\varphi_t$ which do not appear in $t$. For $1 \leq i \leq n$, define $C_i$ to be the singleton $\{c\}$, if $a_i = c$ for some constant $c$, or $\pi_j(Z_k)$, if $a_i = x_j$ for some variable $x_j$. For $1 \leq j \leq k$, define $C_{n+j}$ to be the expression $\pi_{i_j}(Z_k)$, where $x_j$ is the $j$th variable in $\varphi_t$ which does not appear in $t$. Define $q$ to be the query

$$q := \bigcup_{t \in T} \pi_{1, \ldots, n}(\sigma_{\varphi_t}(C_1 \times \cdots \times C_{n+k})).$$
where $\psi_t$ is obtained from $\varphi_t$ by replacing each occurrence of a variable $x_i$ with the index $j$ of the term $C_j$ in which $x_i$ appears. To see that $q(MOD(Z_k)) = MOD(T)$, since $Z_k$ is a c-table, we can use Theorem 4 and check that, in fact, $\bar{q}(Z_k) = T$ where $\bar{q}$ is the translation of $q$ into the c-tables algebra (see the proof of Theorem 4). Note that we only need the $SPJU$ fragment of $RA$.

Example 4. The c-table from Example 2 is definable as $MOD(S) = q(Z_3)$ where $q$ is the following query with input relation name $V$ of arity 3: $q(V) := \pi_{123}(1 \times \{2\} \times V) \cup \pi_{124}(\sigma_{2=3,4\neq 2'}(\{3\} \times V)) \cup \pi_{512}(\sigma_{3\neq 1',3\neq 4}(\{4\} \times \{5\} \times V))$.

Remark 1. It turns out that the i-databases representable by c-tables are also definable via $RA$ starting from the absolute zero-information instance, $N$. Indeed, it can be shown (Proposition 4) that for each $k$ there exists an $RA$ query $q$ such that $Z_k = q(N)$. From there we can apply Theorem 1. The class of incomplete databases $\{I \mid \exists q \in RA \text{ s.t. } I = q(N)\}$ is strictly larger than that representable by c-tables, but it is still countable hence strictly smaller than that of all incomplete databases. Its connections with FO-definability in finite model theory might be interesting to investigate.

Hence, c-tables are in some sense “no more powerful” than the relational algebra. But are they “as powerful”? This justifies the following:

Definition 4. A representation system is $RA$-complete if it can represent any $RA$-definable i-database.

Since $Z_k$ is itself a c-table the following is an immediate corollary of the fundamental result of [20] (see Theorem 4 below). It also states that the converse of Theorem 1 holds.

Theorem 2. c-tables are $RA$-complete.

This result is similar in nature to Corollary 3.1 in [18]. However, the exact technical connection, if any, is unclear, since Corollary 3.1 in [18] relies on the certain answers semantics for queries.

We now turn to the kind of completeness considered in [29].

Definition 5. A representation system is finitely complete if it can represent any finite i-database.

The finite incompleteness of $?-tables, or-set-tables, or-set-?-tables and other systems is discussed in [29] where a finitely complete representation system $RA_{prop}$ is also given (we repeat the definition in the Appendix). Is finite completeness a reasonable question for c-tables, $v$-tables, and Codd tables? In general, for such tables $MOD(T)$ is infinite (all that is needed is a tuple with at least one variable and with an infinitely satisfiable condition). To facilitate comparison with the systems in [29] we define finite-domain versions of tables with variables.

Definition 6. A finite-domain c-table ($v$-table, Codd table) consists of a c-table ($v$-table, Codd table) $T$ together with a finite $\text{dom}(x) \subset \mathbb{D}$ for each variable $x$ that occurs in $T$. 
Note that finite-domain Codd tables are equivalent to or-set tables. Indeed, to obtain an or-set table from a Codd table, one can see $\text{dom}(x)$ as an or-set and substitute it for $x$ in the table. Conversely, to obtain a Codd table from an or-set table, one can substitute a fresh variable $x$ for each or-set and define $\text{dom}(x)$ as the contents of the or-set.

In light of this connection, finite-domain $v$-tables can be thought of as a kind of “correlated” or-set tables. Finite-domain $v$-tables are strictly more expressive than finite Codd tables. Indeed, every finite Codd table is also a finite $v$-table. But, the set of instances represented by e.g. the finite $v$-table $\{(1, x), (x, 1)\}$ where $\text{dom}(x) = \{1, 2\}$ cannot be represented by any finite Codd table. Finite-domain $v$-tables are themselves finitely incomplete. For example, the $i$-database $\{\{(1, 2)\}, \{(2, 1)\}\}$ cannot be represented by any finite $v$-table.

It is easy to see that finite-domain $c$-tables are finitely complete and hence equivalent to $\textbf{R}^4_{\text{prop}}$ in terms of expressive power. In fact, this is true even for the fragment of finite-domain $c$-tables which we will call boolean $c$-tables, where the variables take only boolean values and are only allowed to appear in conditions (never as attribute values).

**Theorem 3.** Boolean $c$-tables are finitely complete (hence finite-domain $c$-tables are also finitely complete).

**Proof.** Let $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ be a finite $i$-database. Construct a boolean $c$-table $T$ such that $\text{Mod}(T) = \mathcal{I}$ as follows. Let $\ell := \lceil \lg m \rceil$. For $1 \leq i < m$, put all the tuples from $I_i$ into $T$ with condition $\varphi_i$, defined

$$\varphi_i := \bigwedge_j \neg x_j \land \bigwedge_k x_k,$$

where the first conjunction is over all $1 \leq j \leq \ell$ such that $j$th digit in the $\ell$-digit binary representation of $i - 1$ is 0, and the second conjunction is over all $1 \leq k \leq \ell$ such that the $k$th digit in the $\ell$-digit binary representation of $i - 1$ is 1. Finally, put all the tuples from $I_m$ into $T$ with condition $\varphi_m \lor \cdots \lor \varphi_{2\ell}$.  

Although boolean $c$-tables are complete there are clear advantages to using variables in tuples also, chief among them being compactness of representations.

**Example 5.** Consider the finite $c$-table $\{(x_1, x_2, \ldots, x_m : \text{true})\}$ where $\text{dom}(x_1) = \text{dom}(x_2) = \cdots = \text{dom}(x_m) = \{1, 2, \ldots, n\}$. The equivalent boolean $c$-table has $n^m$ tuples.

If we additionally restrict boolean $c$-tables to allow conditions to contain only true or a single variable which appears in no other condition, then we obtain a representation system which is equivalent to $?$-tables.

Since finite $c$-tables and $\textbf{R}^4_{\text{prop}}$ are each finitely complete there is an obvious naïve algorithm to translate back and forth between them: list all the instances the one represents, then use the construction from the proof of finite completeness for the other. Finding a more practical “syntactic” algorithm is an interesting open question.
4 Closure Under Relational Operations

**Definition 7.** A representation system is **closed** under a query language if for any query $q$ and any table $T$ there is a table $T'$ that represents $q(\text{Mod}(T))$.

(For notational simplicity we consider only queries with one input relation name, but everything generalizes smoothly to multiple relation names.)

This definition is from [29]. In [2], a **strong** representation system is defined in the same way, with the significant addition that $T'$ should be *computable* from $T$ and $q$. It is not hard to show, using general recursion-theoretic principles, that there exist representation systems (even ones that only represent finite $i$-databases) which are closed as above but not strong in the sense of [2]. However, the concrete systems studied so far are either not closed or if they are closed then the proof provides also the algorithm required by the definition of strong systems. Hence, we see no need to insist upon the distinction.

**Theorem 4 ([20]).** $c$-tables, finite-domain $c$-tables, and boolean $c$-tables are closed under the relational algebra.

**Proof.** (Sketch.) We repeat here the essentials of the proof, including most of the definition of the $c$-table algebra. For each operation $u$ of the relational algebra [20] defines an operation $\bar{u}$ on $c$-tables as follows. For projection, we have

$$\bar{\pi}_\ell(T) := \{ (t' : \varphi_{t'}) \mid t \in T \text{ s.t. } \pi_\ell(t) = t', \varphi_{t'} = \bigvee \varphi_t \}$$

where $\ell$ is a list of indexes and the disjunction is over all $t$ in $T$ such that $\pi_\ell(t) = t'$. For selection, we have

$$\bar{\sigma}_c(T) := \{ (t : \varphi_t \land c(t)) \mid (t, \varphi_t) \in T \}$$

where $c(t)$ denotes the result of evaluating the selection predicate $c$ on the values in $t$ (for a boolean $c$-table, this will always be true or false, while for $c$-tables and finite-domain $c$-tables, this will be in general a boolean formula on constants and variables). For cross product and union, we have

$$T_1 \times T_2 := \{ (t_1 \times t_2 : \varphi_{t_1} \land \varphi_{t_2}) \mid t_1 \in T_1, t_2 \in T_2 \}$$

$$T_1 \cup T_2 := T_1 \cup T_2$$

Difference and intersection are handled similarly. By replacing $u$’s by $\bar{u}$ we translate any relational algebra expression $q$ into a $c$-table algebra expression $\bar{q}$ and it can be shown that

**Lemma 1.** For all valuations $\nu$, $\nu(\bar{q}(T)) = q(\nu(T))$.

From this, $\text{Mod}(\bar{q}(T)) = q(\text{Mod}(T))$ follows immediately. □
5  Algebraic Completion

None of the incomplete representation systems we have seen so far is closed under the full relational algebra. Nor are two more representation systems considered in [29], \( \mathcal{R}_{\text{sets}} \) and \( \mathcal{R}_{\oplus} \) (we repeat their definitions in the Appendix).

**Proposition 1 ([20, 29]).** Codd tables and \( v \)-tables are not closed under e.g. selection. Or-set tables and finite \( v \)-tables are also not closed under e.g. selection. \( \exists \)-tables, \( \mathcal{R}_{\text{sets}} \), and \( \mathcal{R}_{\oplus} \) are not closed under e.g. join.

We have seen that “closing” minimal-information one-row Codd tables (see before Definition 4) \( \{Z_1, Z_2, \ldots\} \), by relational algebra queries yields equivalence with the \( c \)-tables. In this spirit, we will investigate “how much” of the relational algebra would be needed to complete the other representation systems considered. We call this kind of result algebraic completion.

**Definition 8.** If \((\mathcal{T}, \text{Mod})\) is a representation system and \( \mathcal{L} \) is a query language, then the representation system obtained by closing \( \mathcal{T} \) under \( \mathcal{L} \) is the set of tables \( \{(T, q) \mid T \in \mathcal{T}, q \in \mathcal{L}\} \) with the function \( \text{Mod} : \mathcal{T} \times \mathcal{L} \rightarrow \mathcal{N} \) defined by \( \text{Mod}(T, q) := q(\text{Mod}(T)) \).

We are now ready to state our results regarding algebraic completion.

**Theorem 5 (\( \mathcal{RA} \)-Completion).**

1. The representation system obtained by closing Codd tables under SPJU queries is \( \mathcal{RA} \)-complete.
2. The representation system obtained by closing \( v \)-tables under SP queries is \( \mathcal{RA} \)-complete.

**Proof.** (Sketch.) For each case we show that given an arbitrary \( c \)-table \( T \) one can construct a table \( S \) and a query \( q \) of the required type such that \( q(S) = T \). Case 1 is a trivial corollary of Theorem 1. The details for Case 2 are in the Appendix.

Note that in general there may be a “gap” between the language for which closure fails for a representation system and the language required for completion. For example, Codd tables are not closed under selection, but at the same time closing Codd tables under selection does not yield an \( \mathcal{RA} \)-complete representation system. (To see this, consider the incomplete database represented by the \( v \)-table \( \{(x, 1), (x, 2)\} \). Intuitively, selection alone is not powerful enough to yield this incomplete database from a Codd table, as, selection operates on one tuple at a time and cannot correlate two un-correlated tuples.) On the other hand, it is possible that some of the results we present here may be able to be “tightened” to hold for smaller query languages, or else proved to be “tight” already. This is an issue we hope to address in future work.

We give now a set of analogous completion results for the finite case.
**Theorem 6 (Finite-Completion).**

1. The representation system obtained by closing or-set-tables under PJ queries is finitely complete.
2. The representation system obtained by closing finite v-tables under PJ or \(S^+P\) queries is finitely complete.
3. The representation system obtained by closing \(R_{\text{sets}}\) under PJ or PU queries is finitely complete.
4. The representation system obtained by closing \(R_{\oplus\equiv}\) under \(S^+P\) queries is finitely complete.

**Proof.** (Sketch.) In each case, given an arbitrary finite incomplete database, we construct a table and query of the required type which yields the incomplete database. The details are in the Appendix. \(\square\)

Note that there is a gap between the \(R_A\)-completion result for Codd tables, which requires \(SPJU\) queries, and the finite-completion result for finite Codd tables, which requires only \(PJ\) queries. A partial explanation is that proof of the latter result relies essentially on the finiteness of the \(i\)-database.

More generally, if a representation system can represent arbitrarily-large \(i\)-databases, then closing it under \(R_A\) yields a finitely complete representation system, as the following theorem makes precise (see Appendix for proof).

**Theorem 7 (General Finite-Completion).** Let \(T\) be a representation system such that for all \(n \geq 1\) there exists a table \(T\) in \(T\) such that \(|\text{Mod}(T)| \geq n\). Then the representation system obtained by closing \(T\) under \(R_A\) is finitely-complete.

**Corollary 1.** The representation system obtained by closing \(?\)-tables under \(R_A\) queries is finitely complete.

## 6 Probabilistic Databases and Representation Systems

**Finiteness assumption** For the entire discussion of probabilistic database models we will assume that the domain of values \(\mathbb{D}\) is finite. Infinite domains of values are certainly interesting in practice; for some examples see [22, 33, 29]. Moreover, in the case of incomplete databases we have seen that they allow for interesting distinctions.\(^5\) However, finite probability spaces are much simpler than infinite ones and we will take advantage of this simplicity. We leave for future investigations the issues related to probabilistic databases over infinite domains.

We wish to model probabilistic information using a probability space whose possible outcomes are all the conventional instances. Recall that for simplicity we assume a schema consisting of just one relation of arity \(n\). The finiteness of \(\mathbb{D}\) implies that there are only finitely many instances, \(I \subseteq \mathbb{D}^n\).

\(^5\) Note however that the results remain true if \(\mathbb{D}\) is finite; we just require an infinite supply of variables.
By finite probability space we mean a probability space (see e.g. [11]) 
\((\Omega, \mathcal{F}, P)\) in which the set of outcomes \(\Omega\) is finite and the \(\sigma\)-field of events \(\mathcal{F}\) consists of all subsets of \(\Omega\). We shall use the equivalent formulation of pairs 
\((\Omega, p)\) where \(\Omega\) is the finite set of outcomes and where the outcome probability assignment \(p : \Omega \rightarrow [0, 1]\) satisfies \(\sum_{\omega \in \Omega} p(\omega) = 1\). Indeed, we take \(P[A] = \sum_{\omega \in A} p(\omega)\).

**Definition 9.** A probabilistic(-information) database (sometimes called in this paper a \(p\)-database) is a finite probability space whose outcomes are all the conventional instances, i.e., a pair \((\mathcal{N}, p)\) where \(\sum_{I \in \mathcal{N}} p(I) = 1\).

Demanding the direct specification of such probabilistic databases is unrealistic because there are \(2^N\) possible instances, where \(N := |\mathcal{D}|^n\), and we would need that many (minus one) probability values. Thus, as in the case of incomplete databases we define probabilistic representation systems consisting of “probabilistic tables” (prob. tables for short) and a function \(Mod\) that associates to each prob. table \(T\) a probabilistic database \(Mod(T)\). Similarly, we define completeness (finite completeness is the only kind we have in our setting).

To define closure under a query language we face the following problem. Given a probabilistic database \((\mathcal{N}, p)\) and a query \(q\) (with just one input relation name), how do we define the probability assignment for the instances in \(q(\mathcal{N})\)? It turns out that this is a common construction in probability theory: image spaces.

**Definition 10.** Let \((\Omega, p)\) be a finite probability space and let \(f : \Omega \rightarrow \Omega'\) where \(\Omega'\) is some finite set. The image of \((\Omega, p)\) under \(f\) is the finite probability space \((\Omega', p')\) where \(p'(\omega') := \sum_{f(\omega) = \omega'} p(\omega)\).

Again we consider as query languages the relational algebra and its sublanguages defined by subsets of operations.

**Definition 11.** A probabilistic representation system is closed under a query language if for any query \(q\) and any prob. table \(T\) there exists a prob. table \(T'\) that represents \(q(\text{Mod}(T))\), the image space of \(\text{Mod}(T)\) under \(q\).

7 Probabilistic \(p\)-Tables and Probabilistic Or-Set Tables

Probabilistic \(p\)-tables (\(p\)-\(p\)-tables for short) are commonly used for probabilistic models of databases [34,15,16,9] (they are called “independent tuple representation in [30]). Such tables are the probabilistic counterpart of \(p\)-tables where each “?” is replaced by a probability value. Example 6 below shows such a table. The tuples not explicitly shown are assumed tagged with probability 0.

\[\text{Example 6:}\]

\[
\begin{array}{ccc}
P(t_1) & P(t_2) & P(t_3) \\
0.3 & 0.5 & 0.2 \\
\end{array}
\]

In order to represent a probabilistic database, papers using this model typically include a statement like “every tuple \(t\) is in the outcome

---

6 It is easy to check that the \(p'(\omega')\)'s do actually add up to 1.
instance with probability $p_t$, independently from the other tuples” and then a statement like

$$P[I] = \left(\prod_{t \in I} p_t\right)\left(\prod_{t \notin I} (1 - p_t)\right).$$

In fact, to give a rigorous semantics, one needs to define the events $E_t \subseteq N$, $E_t := \{I \mid t \in I\}$ and then to prove the following.

**Proposition 2.** There exists a unique probabilistic database such that the events $E_t$ are jointly independent and $P[E_t] = p_t$.

This defines $p$-?-tables as a probabilistic representation system. We shall however provide an equivalent but more perspicuous definition. We shall need here another common construction from probability theory: product spaces.

**Definition 12.** Let $(\Omega_1, p_1), \ldots, (\Omega_n, p_n)$ be finite probability spaces. Their product is the space $(\Omega_1 \times \cdots \times \Omega_n, p)$ where

$$p(\omega_1, \ldots, \omega_n) := p_1(\omega_1) \cdots p_n(\omega_n).$$

This definition corresponds to the intuition that the $n$ systems or phenomena that are modeled by the spaces $(\Omega_1, p_1), \ldots, (\Omega_n, p_n)$ behave without “interfering” with each other. The following formal statements summarize this intuition.

**Proposition 3.** Consider the product of the spaces $(\Omega_1, p_1), \ldots, (\Omega_n, p_n)$. Let $A_1 \subseteq \Omega_1, \ldots, A_n \subseteq \Omega_n$.

1. We have $P[A_1 \times \cdots \times A_n] = P[A_1] \cdots P[A_n]$.
2. The events $A_1 \times \Omega_2 \times \cdots \times \Omega_n$, $\Omega_1 \times A_2 \times \cdots \times \Omega_n$, $\ldots$, $\Omega_1 \times \Omega_2 \times \cdots \times A_n$ are jointly independent in the product space.

Turning back to $p$-?-tables, for each tuple $t \in \mathbb{D}^n$ consider the finite probability space $B_t := (\{\text{true}, \text{false}\}, p)$ where $p(\text{true}) := p_t$ and $p(\text{false}) = 1 - p_t$. Now consider the product space

$$P := \prod_{t \in \mathbb{D}^n} B_t$$

We can think of its set of outcomes (abusing notation, we will call this set $P$ also) as the set of functions from $\mathbb{D}^n$ to $\{\text{true}, \text{false}\}$, in other words, predicates on $\mathbb{D}^n$. There is an obvious function $f : P \rightarrow N$ that associates to each predicate the set of tuples it maps to $\text{true}$.

All this gives us a $p$-database, namely the image of $P$ under $f$. It remains to show that it satisfies the properties in Proposition 2. Indeed, since $f$ is a bijection, this probabilistic database is in fact isomorphic to $P$. In $P$ the events that are in bijection with the $E_t$’s are the Cartesian product in which there is exactly one component $\{\text{true}\}$ and the rest are $\{\text{true}, \text{false}\}$. The desired properties then follow from Proposition 3.

We define now another simple probabilistic representation system called probabilistic or-set-tables ($p$-or-set-tables for short). These are the probabilistic counterpart of or-set-tables where the attribute values are, instead of
or-sets, finite probability spaces whose outcomes are the values in the or-set. p-or-set-tables correspond to a simplified version of the ProbView model presented in [22], in which plain probability values are used instead of confidence intervals.


$S := \begin{bmatrix} 1 & (2 : 0.3, 3 : 0.7) \\ 4 & 5 \\ (6 : 0.5, 7 : 0.5) & (8 : 0.1, 9 : 0.9) \end{bmatrix}$

$T := \begin{bmatrix} 1 & 2 & 0.4 \\ 3 & 4 & 0.3 \\ 5 & 6 & 1.0 \end{bmatrix}$

A p-or-set-table determines an instance by choosing an outcome in each of the spaces that appear as attribute values, independently. Recall that or-set tables are equivalent to finite-domain Codd tables. Similarly, a p-or-set-table corresponds to a Codd table $T$ plus for each variable $x$ in $T$ a finite probability space $\text{dom}(x)$ whose outcomes are in $\mathbb{D}$. This yields a p-database, again by image space construction, as shown more generally for c-tables next in section 8.

Query answering The papers [15, 34, 22] have considered, independently, the problem of calculating the probability of tuples appearing in query answers. This does not mean that in general $q(\text{Mod}(T))$ can be represented by another tuple table when $T$ is some p-?-table and $q \in \mathcal{RA}$ (neither does this hold for p-or-set-tables). This follows from Proposition 1. Indeed, if the probabilistic counterpart of an incompleteness representation system $T$ is closed, then so is $T$. Hence the lifting of the results in Proposition 1 and other similar results.

Each of the papers [15, 34, 22] recognizes the problem of query answering and solves it by developing a more general model in which rows contain additional information similar in spirit to the conditions that appear in c-tables (in fact [15]'s model is essentially what we call probabilistic boolean c-tables, see next section). We will show that we can actually use a probabilistic counterpart to c-tables themselves together with the algebra on c-tables given in [20] to achieve the same effect.

8 Probabilistic c-tables

Definition 13. A probabilistic c-table (pc-tables for short) consists of a c-table $T$ together with a finite probability space $\text{dom}(x)$ (whose outcomes are values in $\mathbb{D}$) for each variable $x$ that occurs in $T$.

To get a probabilistic representation system consider the product space

$$V := \prod_{x \in \text{Var}(T)} \text{dom}(x)$$

The outcomes of this space are in fact the valuations for the c-table $T$! Hence we can define the function $g : V \rightarrow \mathcal{N}, g(\nu) := \nu(T)$ and then define $\text{Mod}(T)$ as the image of $V$ under $g$.

Similarly, we can talk about boolean pc-tables, pv-tables and probabilistic Codd tables (the latter related to [22], see previous section). Moreover, the p-?-tables correspond to restricted boolean pc-tables, just like ?-tables.
Theorem 8. Boolean pc-tables are complete (hence pc-tables are also complete).

Proof. Let $I_1, \ldots, I_k$ denote the instances with non-zero probability in an arbitrary probabilistic database, and let $p_1, \ldots, p_k$ denote their probabilities. Construct a probabilistic boolean $c$-table $T$ as follows. For $1 \leq i \leq k - 1$, put the tuples from $I_i$ in $T$ with condition $\neg x_1 \land \cdots \land \neg x_{i-1} \land x_i$. Put the tuples from $I_k$ in $T$ with condition $\neg x_1 \land \cdots \land \neg x_{k-1}$. For $1 \leq i \leq k - 1$, set $\mathbb{P}[x_i = \text{true}] := p_i / (1 - \sum_{j=1}^{i-1} p_j)$. It is straightforward to check that this yields a table such that $\mathbb{P}[I_i] = p_i$. \hfill $\square$

The previous theorem was independently observed in [30].

Theorem 9. pc-tables (and boolean pc-tables) are closed under the relational algebra.

Proof. (Sketch.) For any pc-table $T$ and any $\mathcal{RA}$ query $q$ we show that the probability space $q(\text{Mod}(T))$ (the image of $\text{Mod}(T)$ under $q$) is in fact the same as the space $\text{Mod}(\overline{q}(T))$. The proof of Theorem 4 already shows that the outcomes of the two spaces are the same. The fact that the probabilities assigned to each outcome are the same follows from Lemma 1. \hfill $\square$

The proof of this theorem gives in fact an algorithm for constructing the answer as a $p$-database itself, represented by a pc-table. In particular this will work for the models of [15, 22, 34] or for models we might invent by adding probabilistic information to $v$-tables or to the representation systems considered in [29]. The interesting result of [9] about the applicability of an “extensional” algorithm to calculating answer tuple probabilities can be seen also as characterizing the conjunctive queries $q$ which for any $p$-?-table $T$ are such that the $c$-table $\overline{q}(T)$ is in fact equivalent to some $p$-?-table.

9 Some Ideas for Further Work

The new results on algebraic completion may not be as tight as they can be. Ideally, we would like to be able show that for each representation system we consider, the fragment of $\mathcal{RA}$ we use is minimal in the sense that closing the representation system under a more restricted fragment does not obtain a complete representation system.

We did not consider $c$-tables with global conditions [17] nor did we describe the exact connection to logical databases [27, 32]. Even more importantly, we did not consider complexity issues as in [3]. All of the above are important topics for further work, especially the complexity issues and the related issues of succinctness/compactness of the table representations.

As we see, in pc-tables the probability distribution is on the values taken by the variables that occur in the table. The variables are assumed independent here. This is a lot more flexible (as the example shows) than independent tuples, but still debatable. Consequently, as part of the proposed work, trying to make
pc-tables even more flexible, we plan to investigate models in which the assumption that the variables take values independently is relaxed by using conditional probability distributions [14].

Space limitations prevent us from giving details, but there is a good reason why the c-table algebra was in essence rediscovered in [15, 22, 34] and to some extent in [28]. The condition that decorates a tuple \( t \) in \( q(T) \) can be seen as the lineage [8], a.k.a. the why-provenance [6], of the tuple \( t \). We plan to discuss elsewhere the connection between algorithms for computing why-provenance and the c-table algebra.

It would be interesting to connect this work to the extensive literature on disjunctive databases, see e.g., [24], and to the work on probabilistic object-oriented databases [12].

Probabilistic modeling is by no means the only way to model uncertainty in information systems. In particular it would be interesting to investigate possibility models [19] for databases, perhaps following again, as we did here, the parallel with incompleteness.

References

Appendix

Proposition 4. There exists a relational query $q$ such that $q(N) = Z_n$.

Proof. Define sub-query $q'$ to be the relational query

$$q'(V) := V - \pi_{\ell}(\sigma_{r \neq r}(V \times V)),$$

where $\ell$ is short for $1, \ldots, n$ and $\ell \neq r$ is short for $1 \neq n + 1 \lor \cdots \lor n \neq 2n$. Note that $q'$ yields $V$ if $V$ consists of a single tuple and $\emptyset$ otherwise. Now define $q$ to be the relational query

$$q(V) := q'(V) \cup (\{t\} - \pi_\ell(\{t\} \times q'(V)))$$

where $t$ is a tuple chosen arbitrarily from $D^n$. It is clear that $q(N) = Z_n$.

**Definition 14.** A table in the representation system $R_{sets}$ is a multiset of sets of tuples, or blocks, each such block optionally labeled with a ‘?’ if $T$ is an $R_{sets}$ table, then $\text{Mod}(T)$ is the set of instances obtained by choosing one tuple from each block not labeled with a ‘?’ and at most one tuple from each block labeled with a ‘?’.

**Definition 15.** A table in the representation system $R_{\oplus \equiv}$ is a multiset of tuples $\{t_1, \ldots, t_m\}$ and a conjunction of logical assertions of the form $i \oplus j$ (meaning $t_i$ or $t_j$ must be present in an instance, but not both) or $i \equiv j$ (meaning $t_i$ is present in an instance iff $t_j$ is present in the instance). If $T$ is an $R_{\oplus \equiv}$ table then $\text{Mod}(T)$ consists of all subsets of the tuples satisfying the conjunction of assertions.

**Definition 16.** A table in the representation system $R_{A\text{-prop}}$ is a multiset of or-set tuples $\{t_1, \ldots, t_m\}$ and a boolean formula on the variables $\{t_1, \ldots, t_m\}$. If $T$ is an $R_{A\text{-prop}}$ table then $\text{Mod}(T)$ consists of all subsets of the tuples satisfying the boolean assertion, where the variable $t_i$ has value true iff the tuple $t_i$ is present in the subset.

**Theorem 5 (RA-Completion).**

1. The representation system obtained by closing Codd tables under $SPJU$ queries is $RA$-complete.
2. The representation system obtained by closing $v$-tables under $SP$ queries is $RA$-complete.

**Proof.** In each case we show that given an arbitrary $c$-table $T$, one can construct a table $S$ and a query $q$ such that $q(S) = T$.

1. Trivial corollary of Theorem 1.
2. Let $k$ be the arity of $T$. Let $\{t_1, \ldots, t_m\}$ be an enumeration of the tuples of $T$, and let $\{x_1, \ldots, x_n\}$ be an enumeration of the variables which appear in $T$. Construct a $v$-table $S$ with arity $k + n + 1$ as follows. For every tuple $t_i$ in $T$, put exactly one tuple $t'_i$ in $S$, where $t'_i$ agrees with $t_i$ on the first $k$ columns, the $k + 1$st column contains the constant $i$, and the last $m$ columns contain the variables $x_1, \ldots, x_m$. Now let $q$ be the $SP$ query defined

$$q := \pi_1, \ldots, k(\sigma_{i=1}^{n}(k+1=i) \land \psi_i(S))$$

where $\psi_i$ is obtained from the condition $\varphi_i$ of tuple $t_i$ by replacing variable names with their corresponding indexes in $S$. 

64
Theorem 6 (Finite-Completion).

1. The representation system obtained by closing or-set-tables under PJ queries is finitely complete.
2. The representation system obtained by closing finite v-tables under PJ or S+P queries is finitely complete.
3. The representation system obtained by closing R sets under PJ or PU queries is finitely complete.
4. The representation system obtained by closing R≡ under S+PJ queries is finitely complete.

Proof. Fix an arbitrary finite incomplete database \( I = \{I_1, \ldots, I_n\} \) of arity \( k \).
It suffices to show in each case that one can construct a table \( T \) in the given representation system and a query \( q \) in the given language such that \( q(\text{Mod}(T)) = I \).

1. We construct a pair of or-set-tables \( S \) and \( T \) as follows. (They can be combined together into a single table, but we keep them separate to simplify the presentation.) For each instance \( I_i \) in \( I \), we put all the tuples of \( I_i \) in \( S \), appending an extra column containing value \( i \). Let \( T \) be the or-set-table of arity 1 containing a single tuple whose single value is the or-set \( \langle 1, 2, \ldots, n \rangle \).

Now let \( q \) be the \( S+P \) query defined:

\[
q := \pi_{1, \ldots, k} \sigma_{k+1 = k+2}(S \times T).
\]

2. Completion for PJ follows from Case 1 and the fact that finite v-tables are strictly more expressive than or-set tables. For \( S+P \), take the finite v-table representing the cross product of \( S \) and \( T \) in the construction from Case 1, and let \( q \) be the obvious \( S+P \) query.

3. Completion for PJ follows from Case 1 and the fact (shown in [29]) that or-set-tables are strictly less expressive than R sets. Thus we just need show the construction for PU. We construct an R sets table \( T \) as follows. Let \( m \) be the cardinality of the largest instance in \( I \). Then \( T \) will have arity \( km \) and will consist of a single block of tuples. For every instance \( I_i \) in \( I \), we put one tuple in \( T \) which has every tuple from \( I_i \) arranged in a row. (If the cardinality of \( I_i \) is less than \( m \), we pad the remainder with arbitrary tuples from \( I_i \).) Now let \( q \) be the PU query defined as follows:

\[
q := \bigcup_{i=0}^{m-1} \pi_{k_i, \ldots, k_i+k-1}(T).
\]

4. We construct a pair of R≡-tables \( S \) and \( T \) as follows. (\( S \) can be encoded as a special tuple in \( T \), but we keep it separate to simplify the presentation.) Let \( m = \lceil \lg n \rceil \). \( T \) is constructed as in Case 2. \( S \) is a binary table containing,
for each \( i, 1 \leq i \leq m \), a pair of tuples \((0, i)\) and \((1, i)\) with an exclusive-or constraint between them. Let sub-query \( q' \) be defined

\[
q' := \prod_{i=1}^{m} \pi_1(\sigma_{\sigma_2=i}(S))
\]

The \( S^+ PJ \) query \( q \) is defined as in Case 2, but using this definition of \( q' \).

\( \square \)

**Theorem 7 (GeneralFinite Completion).** Let \( T \) be a representation system such that for all \( n \geq 1 \) there exists a table \( T \in T \) such that \( |\text{Mod}(T)| \geq n \). Then the representation system obtained by closing \( T \) under \( RA \) is finitely-complete.

**Proof.** Let \( T \) be a representation system such that for all \( n \geq 1 \) there is a table \( T \in T \) such that \( |\text{Mod}(T)| \geq n \). Let \( \mathcal{I} = \{I_1, \ldots, I_k\} \) be an arbitrary non-empty finite set of instances of arity \( m \). Let \( T \) be a table in \( T \) such that \( \text{Mod}(T) = \{J_1, \ldots, J_\ell\} \), with \( \ell \geq k \). Define \( RA \) query \( q \) to be

\[
q(V) := \bigcup_{1 \leq i \leq k-1} I_i \times q_i(V) \cup \bigcup_{k \leq i \leq \ell} I_k \times q_i(V),
\]

where \( I_i \) is the query which constructs instance \( I_i \) and \( q_i(V) \) is the boolean query which returns true iff \( V \) is identical to \( I_i \) (which can be done in \( RA \)). Then \( q(\text{Mod}(T)) = \mathcal{I} \).

\( \square \)
Preference-Driven Querying of Inconsistent Relational Databases *

Slawomir Staworko, Jan Chomicki, and Jerzy Marcinkowski

1 University at Buffalo, {staworko,chomicki}@cse.buffalo.edu
2 Wroclaw University
Jerzy.Marcinkowski@ii.uni.wroc.pl

Abstract. One of the goals of cleaning an inconsistent database is to remove conflicts between tuples. Typically, the user specifies how the conflicts should be resolved. Sometimes this specification is incomplete, and the cleaned database may still be inconsistent. At the same time, data cleaning is a rather drastic approach to conflict resolution: It removes tuples from the database, which may lead to information loss and inaccurate query answers.

We investigate an approach which constitutes an alternative to data cleaning. The approach incorporates preference-driven conflict resolution into query answering. The database is not changed. These goals are achieved by augmenting the framework of consistent query answers through various notions of preferred repair. We axiomatize desirable properties of preferred repair families and propose different notions of repair optimality. Finally, we investigate the computational complexity implications of introducing preferences into the computation of consistent query answers.

1 Introduction

In many novel database applications, violations of integrity constraints cannot be avoided. A typical example is integration of two consistent data sources that contribute conflicting information. At the same time the sources are autonomous and cannot be changed. Inconsistencies also occur in the context of long running operations. Finally, integrity enforcement may be neglected because of efficiency considerations.

Integrity constraints, however, often capture important semantic properties of the stored data. These properties directly influence the way a user formulates a query. Evaluation of the query over an inconsistent database may negatively affect the meaning of the answers.

Example 1. Consider the schema

$$ \text{Mgr}(\text{Name}, \text{Dept}, \text{Salary}, \text{Reports}) $$

consistent with two key dependencies:

$$ \text{Dept} \rightarrow \text{NameSalaryReports}, \quad (fd_1) $$

$$ \text{Name} \rightarrow \text{DeptSalaryReports}, \quad (fd_2) $$

* Research supported by NSF Grants IIS-0119186 and IIS-0307434.
In an instance of this schema a tuple \((x, y, z, v)\) denotes a manager \(x\) of the department \(y\) with the salary \(z\) required to write \(v\) reports annually.

Now suppose we integrate the following (consistent) sources:

\[
s_1 = \{(Mary, R&D, 40k, 3)\}, \quad s_2 = \{(John, R&D, 10k, 2)\}, \quad s_3 = \{(Mary, IT, 20k, 1), (John, PR, 30k, 4)\}.
\]

The integrated instance \(r = s_1 \cup s_2 \cup s_3\) contains 3 conflicts:

1. \((Mary, R&D, 40k, 3)\) and \((John, R&D, 10k, 2)\) w.r.t. \(fd_1\),
2. \((Mary, R&D, 40k, 3)\) and \((Mary, IT, 20k, 1)\) w.r.t. \(fd_2\),
3. \((John, R&D, 10k, 2)\) and \((John, PR, 30k, 4)\) w.r.t. \(fd_2\).

These inconsistencies can be a result of changes that are not yet fully propagated. For example **Mary** may have been promoted to manage **R&D** whose previous manager **John** was moved to manage **PR**, or conversely **John** may have been moved to manage **R&D**, while **Mary** was moved from **R&D** to manage **IT**.

Consider the query \(Q_1\) asking if **John** earns more than **Mary**:

\[
\exists x_1, y_1, z_1, x_2, y_2, z_2. Mgr(Mary, x_1, y_1, z_1) \land Mgr(John, x_2, y_2, z_2) \land y_1 < y_2.
\]

The answer to \(Q_1\) in \(r\) is true but this is misleading because \(r\) may not correspond to any actual state of the world.

One way to deal with the impact of inconsistencies in the results of the query evaluation is **data cleaning** [16]. Although there exist a wide variety of tools for automatic elimination of duplicates, extraction and standardization of information, there are practically no tools that automatically resolve integrity constraint violations [18]. Usually, the user is responsible for providing a procedure that decides how the conflicts should be resolved. The standard repertoire of actions that can be performed on a conflicting tuple is [23]: removing the tuple, leaving the tuple, or reporting the tuple to an auxiliary (**contingency**) table. Typically, the data cleaning system provides useful information which may include:

- the timestamp of creation/last modification of the tuple (the conflicts can be resolved by removing from consideration old, outdated tuples),
- source of the information of the tuple (a user can consider the data from one source more reliable than the data from the other).

The approach of data cleaning has several shortcomings:

- If the user provides insufficient information to resolve all the conflicts then data cleaning results in an inconsistent database; this again may lead to misleading answers.
- Physically removing the tuples from the database may lead to information loss.
- Data cleaning doesn’t allow to utilize the incomplete information often expressed in inconsistencies.
The framework of repairs and consistent query answers [1] proposes an alternative approach to deal with inconsistent databases geared towards utilizing incomplete information. A repair is a minimally changed consistent database and a consistent answer to a query is the answer present in every repair. This approach doesn’t remove physically any tuples from the database. The framework of [1] has served as a foundation for most of the subsequent work in the area of querying inconsistent databases (for recent developments see [3, 11]).

Example 2. The instance r of Example 1 has 3 repairs:

\[ r_1 = \{ (Mary, R&D, 40k, 3), (John, PR, 30k, 4) \}, \]
\[ r_2 = \{ (John, R&D, 10k, 3), (Mary, IT, 20k, 1) \}, \]
\[ r_3 = \{ (Mary, IT, 20k, 1), (John, PR, 30k, 4) \}. \]

Because \( Q_1 \) is false in \( r_1 \) and \( r_2 \), true is not a consistent answer to \( Q_1 \).

The standard framework of consistent query answers does not contain any way to incorporate additional user input about how to resolve some conflicts. One can try to first clean the database and then use the consistent query answers approach. This is a radical approach: removing tuples may lead to information loss. Additional user input in the form of preferences can be used in the framework of consistent query answers to benefit the correctness of consistent query answers by considering only the preferred repairs.

Example 3. Suppose the user finds the source \( s_3 \) to be less reliable than \( s_1 \) and less reliable than \( s_2 \). The user does not know, however, the relative reliability of the sources \( s_1 \) and \( s_2 \). The cleaning of \( r \) with this information yields an inconsistent database:

\[ r' = \{ (Mary, R&D, 40k, 3), (John, R&D, 10k, 2) \}. \]

Consider the query \( Q_2 \) asking if Mary earns more and has fewer reports to write than John:

\[ \exists x_1, y_1, z_1, x_2, y_2, z_2. Mgr(Mary, x_1, y_1, z_1) \land Mgr(John, x_2, y_2, z_2) \land y_1 > y_2 \land z_1 < z_2. \]

The answer to this query in the “cleaned” database is false. False is also the consistent answer to \( Q_2 \) in \( r' \). Note, however, that neither false nor true is a consistent answer to \( Q_2 \) in \( r \).

Intuitively the repairs \( r_1 \) and \( r_2 \) incorporate more of reliable information than the repair \( r_3 \) (all tuples of \( r_3 \) come from a less reliable source \( s_3 \)). If we consider \( r_1 \) and \( r_2 \) at the only preferred repairs, then true is the preferred consistent answer to \( Q_2 \).

In our paper we extend the framework of consistent query answers with an additional input consisting of preference information \( \Phi \). We use \( \Phi \) to define the set of preferred repairs \( Rep^\Phi \). When we compute consistent answers, instead of considering the set of all repairs \( Rep \), we use the set of preferred repairs. We assume that there exists a (possibly partial) operation of extending \( \Phi \) with some additional preference information and we write \( \Phi \subseteq \Psi \) when \( \Psi \) is an extension of \( \Phi \). We consider \( \Phi \) to be total when it cannot be extended further. We identify the following desirable properties of families of preferred repairs:
1. **Non-emptiness**
   \[ \text{Rep}^\Phi \neq \emptyset. \]  
   \((\mathcal{P}1)\)

2. **Monotonicity**: extending preferences can only narrow the set of preferred repairs
   \[ \Phi \subseteq \Psi \Rightarrow \text{Rep}^\Psi \subseteq \text{Rep}^\Phi. \]  
   \((\mathcal{P}2)\)

3. **Non-discrimination**: if no preference information is given, then no repair is removed from consideration
   \[ \text{Rep}^\emptyset = \text{Rep}. \]  
   \((\mathcal{P}3)\)

4. **Categoricity**: given maximal preference information we obtain exactly one repair
   \[ \Phi \text{ is total } \Rightarrow |\text{Rep}^\Phi| = 1. \]  
   \((\mathcal{P}4)\)

In Section 3 we observe, however, that these properties do not enforce practically any use of preference information. To do so we also study different notions of repair optimality which ensure a proper use of preference information to select preferred repairs.

## 2 Preliminaries

In this paper, we work with databases over a schema consisting of only one relation \( R \) with attributes from \( U \). We use \( A, B, \ldots \) to denote elements of \( U \) and \( X, Y, \ldots \) to denote subsets of \( U \). We consider two disjoint domains: uninterpreted names \( D \) and natural numbers \( N \). Every attribute in \( U \) is typed. We assume that constants with different names are different and that symbols \( =, \neq, <, > \) have the natural interpretation over \( N \).

The instances of \( R \), denoted by \( r, r', \ldots \), can be seen as finite, first-order structures, that share the domains \( D \) and \( N \). For any tuple \( t \) from \( r \) by \( t.A \) we denote the value associated with the attribute \( A \). In this paper we consider first-order queries over the alphabet consisting of \( \mathcal{R} \) and binary relation symbols \( =, \neq, <, > \).

The limitation to only one relation is made only for the sake of clarity and along the lines of \([7]\) the framework can be easily extended to handle databases with multiple relations.

### 2.1 Inconsistency and repairs

The class of integrity constraints we study consists of functional dependencies. We use \( X \rightarrow Y \) to denote the following constraint:

\[
\forall t_1, t_2 \in R. \bigwedge_{A \in X} t_1.A = t_2.A \Rightarrow \bigwedge_{B \in Y} t_1.B = t_2.B
\]

We identify conflicts created by (1) as follows: tuples \( t_1 \) and \( t_2 \) are **conflicting** if \( t_1.A = t_2.A \) for all \( A \in X \) and \( t_1.B \neq t_2.B \) for some \( B \in Y \). A database \( r \) is **inconsistent** with a set of constraints \( F \) if and only if \( r \) contains some conflicting tuples with a constraint from \( F \). Otherwise, the database is **consistent**.

In the general framework when repairing a database we consider two operations: adding or removing a tuple. Because in the presence of functional dependencies adding new tuples cannot remove conflicts, we only consider repairs obtained by deleting tuples from the original instance.
**Definition 1 (Repair).** Given a database $r$ and a set of integrity constraints $F$, a database $r'$ is a repair of $r$ w.r.t. $F$ if $r'$ is a maximal subset of $r$ consistent with $F$. By $\text{Rep}_F(r)$ we denote the set of all repairs of $r$ w.r.t $F$.

A repair can be viewed as the result of a process of cleaning the input relation. Note that since every conflict can be resolved in two different ways and conflict are often independent, there may be an exponential number of repairs.

**Example 4.** For any natural number $n$ consider an instance

$$r_n = \{(0,0),(0,1),\ldots,(n-1,0),(n-1,1)\}$$

of the schema $R(A,B)$. Note that the set of all repairs of $r_n$ w.r.t. the functional dependency $A \rightarrow B$ is equal to the set $\{0,1\}^n$ of all functions from $\{0,\ldots,n-1\}$ to $\{0,1\}$.

Also note that the set of repairs of a consistent relation $r$ contains only $r$.

Given a relation instance $r$ and a set of functional dependencies $F$, a conflict graph is a graph whose vertices are the tuples of $r$ and two tuples are adjacent only if they are conflicting w.r.t. a constraint from $F$. Conflict graphs are compact representations of repairs because the set of all repairs is equal to the set of all maximal sets of the corresponding conflict graph.

**Example 5.** The conflict graph for the instance $r_n$ for $n = 4$ and the functional dependency $A \rightarrow B$ from Example 4 is presented in Figure 1.

```
(0,1) (1,1) (2,1) (3,1)
\|     \|     \|     \|
(0,0) (1,0) (2,0) (3,0)
```

**Fig. 1.** A conflict graph.

For a given tuple $t$, by $n(t)$ we denote its neighborhood in the conflict graph, i.e. all tuples conflicting with $t$; and the vicinity of $t$ is $v(t) = \{t\} \cup n(t)$.

### 2.2 Priorities and preferred repairs

For the clarity of presentation we assume a fixed database instance $r$ with a fixed set of functional dependencies $F$.

To represent the preference information, we use (possibly partial) acyclic orientations of the conflict graph. Orientations allows us to express the preferences at the level of single conflicts and acyclicity ensures unambiguity of the preference.
Definition 2 (Priority). A priority is a binary relation $\succ \subseteq r \times r$ that is defined only on conflicting tuples and is acyclic, i.e. there does not exist $x \in r$ such that $x \succ^* x$, where $\succ^*$ is the transitive closure of $\succ$. If $x \succ y$ we say that that $x$ dominates $y$. A priority $\succ$ is total if every pair of conflicting tuples $\{x, y\}$ either $x \prec y$ or $y \prec x$.

From the point of user interface it is often more natural to define the priority as an arbitrary acyclic binary relation on $r$ and then use such a priority relation only on conflicting tuples. Naturally, those approaches are equivalent.

Extending an orientation consists of orienting some conflicting edges that were not oriented before; formally, a priority $\succ'$ is an extension of $\succ$ if $\succ' \supseteq \succ$. Note that an extension $\succ'$ is also a priority and therefore $\succ'$ is acyclic and defined only on pairs of conflicting tuples. Also observe that a priority that cannot be extended further is total (i.e. all edges of the conflict graph are oriented).

Preferred repairs In our work we investigate families of preferred repairs: subsets of repairs selected with priorities. For the clarity we adapt the following naming convention. For each investigated way of selecting preferred repairs we use one letter name to refer to it, e.g. $X$. For a given relation $r$, a given set of functional dependencies $F$ and a given priority $\succ$, by $X\text{-Rep}^*_{\succ}(r)$ we denote the selected set of preferred repairs. We drop $r$, $F$, and $\succ$ if they are known from the context.

Database cleaning A total priority represent an unambiguous information on how each conflict should be resolved. With Algorithm 1 a total priority is used to construct a “clean” database by iteratively selecting tuples that are not dominated by any other, i.e. tuples selected by the winnow operator [5]:

$$\omega_\succ (r) = \{ t \in r | \neg \exists t' \in r. t' \succ t \}.$$

After selecting a tuple we remove it and its neighbors from further considerations.

Algorithm 1 Cleaning the database

1: $r' \leftarrow \emptyset$
2: while $\omega_\succ (r) \neq \emptyset$ do
3: choose any $x \in \omega_\succ (r)$
4: $r' \leftarrow r' \cup \{x\}$
5: $r \leftarrow r \setminus (\{x\} \cup n(x))$ $\triangleright$ where $n(x)$ – the neighborhood of $x$.
6: return $r'$

Proposition 1. For a total priority $\succ$ Algorithm 1 computes a unique repair for any sequence of choices in Step 3.

2.3 Preferred consistent query answers

We generalize the notion of consistent query answer [1] by considering only preferred repairs when evaluating a query (instead of all repairs). We only study closed first-order
logic queries. We can easily generalize our approach to open queries along the lines of [1, 7]. For a given query $Q$ we say that true is an answer to $Q$ in $r$, if $r = Q$ in the standard model-theoretic sense.

**Definition 3 (X-Consistent query answer).** Given a closed query $Q$ and a family of repairs $X$-Rep, true is the $X$-consistent query answer to a query $Q$ if for every repair $r' \in X$-Rep we have $r' = Q$.

Note that we obtain the original notion of consistent query answer [1] if we consider the whole set of repairs $Rep_F(r)$.

3 Optimal use of the priority

The main purpose of introducing $P_1$–$P_4$ is identification of desired properties of families of preferred repairs. We note that all properties except for $P_4$ do not require any use of the priority to eliminate any repairs. This makes it possible to construct a family of preferred repairs which satisfies $P_1$–$P_4$ which practically makes no use of the given priority.

**Example 6.** Consider a family of repairs which for a total priority consists of the clean database obtained with Algorithm 1 and otherwise it consists of all repairs. This family of repairs fulfills properties $P_1$–$P_4$.

Thus we investigate a number of increasingly complex notions of repair optimality that ensure effective use of the preference information:

1. $r'$ is a locally optimal repair, if no tuple $x$ from $r'$ can be replaced with a tuple $y$ such that $y \succ x$ and the resulting set of tuples is consistent;
2. $r'$ is a semi-globally optimal if no nonempty subset $X$ of tuples from $r'$ can be replaced with a tuple $y$ such that $\forall x \in X. y \succ x$ and the resulting set of tuples is consistent;
3. $r'$ is a globally optimal if no nonempty subset $X$ of tuples from $r$ can be replaced with a set of tuples $Y$ such that $\forall x \in X. \exists y \in Y. y \succ x$ and the resulting set of tuples is consistent.

We note that global optimality implies semi-global optimality which in turn implies local optimality. Intuitively, global optimality makes an aggressive use of priorities to select repairs, while local optimality does so in a less aggressive manner.

3.1 Locally optimal repairs

With $L$-Rep we denote the set of all locally optimal repairs. The following example illustrates that the notion of local optimality allows to effectively use priorities to handle relations with one key.

**Example 7.** Consider the relational schema $R(A,B)$ with a key dependency $F = \{A \rightarrow B\}$ and take an instance $r = \{t_a = (1, 1), t_b = (1, 2), t_c = (1, 3)\}$ with the priority $\succ = \{(t_a, t_c), (t_a, t_b)\}$. Figure 2 contains the corresponding conflict graph and its orientation. The repairs are $Rep_F(r) = \{r_1 = \{t_a\}, r_2 = \{t_b\}, r_3 = \{t_c\}\}$. Only $r_1$ is locally preferred.
Proposition 2. $L$-Rep satisfies properties $\mathcal{P}_1$–$\mathcal{P}_3$.

As it’s shown on the following example, locally optimal repairs do not satisfy $\mathcal{P}_4$.

Example 8. Consider the relational schema $R(A,B,C)$ with a functional dependency $A \rightarrow B$ and take an instance $r = \{t_a = (1,1,1), t_b = (1,1,2), t_c = (1,2,3)\}$ with the total priority $\succ = \{ (t_c, t_a), (t_c, t_b) \}$. The corresponding conflict graph can be found in Figure 3. The set of repairs consists of two repairs $Rep_F(r) = \{ r_1 = \{ t_a, t_b \}, r_2 = \{ t_c \} \}$. All the repairs are locally optimal.

3.2 Semi-globally optimal repairs

In Example 8, we note that even though the priority suggest rejecting $r_1$ from consideration, the notion of local optimality is too weak to do so. The main reason is the existence of violations of functional dependency with duplicates ($t_a$ and $t_b$ which are not conflicting, but both of them conflict with $t_c$). The notion of semi-global optimality, however, effectively applies the priority in the situations of violations of one non-key functional dependency: the repair $r_1$ is not semi-globally optimal and $r_2$ is. We denote the family of all semi-globally optimal repairs by $S$-Rep and we note that $S$-Rep is as effective in enforcing priorities as $L$-Rep.


Also this family of preferred repairs does not satisfy $\mathcal{P}_4$.

Example 9. Consider the schema $R(A,B,C,D)$ with two functional dependencies $F = \{ A \rightarrow B, C \rightarrow D \}$ and suppose we have a database: $r = \{ t_a = (1,1,0,0), t_b = (1,2,1,1), t_c = (2,1,1,2), t_d = (2,2,2,1), t_e = (0,0,2,2)\}$ with a total priority $\succ = \{ (t_a, t_b), (t_b, t_c), (t_c, t_d), (t_d, t_e) \}$. The conflict graph is presented on Figure 4. The set of repairs is $Rep_F(r) = \{ r_1 = \{ t_a, t_c, t_e \}, r_2 = \{ t_b, t_d \} \}$. This is also the set of semi-globally optimal repairs.

3.3 Globally optimal repairs

Situations similar to Example 9 are encountered in the setting where a relation has more than one functional dependency which are violated by mutual conflicts (a tuple
may be involved in conflicts generated by more than one functional dependency) and
the user provides priority only for some of the violated functional dependencies. In
those settings the notion of global optimality follows our intuitions: \( r_2 \) is not globally
optimal and \( r_1 \) is.

Let \( \mathcal{G} \)-Rep be the family of globally optimal repairs. This family satisfies \( \mathcal{P} 4 \).

**Proposition 4.** \( \mathcal{G} \)-Rep satisfies properties \( \mathcal{P} 1 \)–\( \mathcal{P} 4 \). Moreover \( \mathcal{G} \)-Rep \( \subseteq \) \( S \)-Rep and for

one functional dependency \( \mathcal{G} \)-Rep coincides with \( S \)-Rep.

Globally optimal repairs can be characterized in an alternative way.

**Proposition 5.** For a given priority \( \succ \) and two repairs, we say that \( r_2 \) is preferred over

\( r_1 \), denoted \( r_1 \ll r_2 \), if

\[
\forall x \in r_1 \setminus r_2 . \exists y \in r_2 \setminus r_1 . y \succ x.
\]

A repair \( r' \) is globally optimal if and only if it’s \( \ll \)-maximal (there is no repair \( r'' \) such

that \( r' \ll r'' \)).

This particular “lifting” of a preference on objects to a preference on sets of objects can

be found in other contexts. For example, a similar definition is used for a preference

among different models of a logic program [21], or for a preference among different

worlds [15].

### 3.4 Importance of monotonicity

In Section 4 we study the computational implications of using priorities to handle in-

consistent databases. Restricting our choice when constructing a family of repairs to one

of the optimal classes of repairs, still does not prevent us to construct trivial families of

optimal repairs.

**Example 10.** For any instance \( r \), any set of functional dependencies \( F \), and any priority

\( \succ \) for \( r \) and \( F \), choose one extension \( \succ' \) that is total for \( r \) and \( F \). Now, consider the family
\( T\)-Rep which for an instance \( r \), a set of functional dependencies \( F \), and a priority \( \succ \)

consists of the only repair constructed with Algorithm 1 for \( r, F \), and the corresponding

total priority \( \succ' \).

We can easily show that the repair obtained with Algorithm 1 for a total priority is

a globally optimal repair. Therefore \( T\)-Rep is a family of globally optimal repairs that

satisfies \( \mathcal{P} 1 \), \( \mathcal{P} 3 \), and \( \mathcal{P} 4 \).

We conclude here that while optimality enforces use of priorities to eliminate repairs

from considerations, the monotonicity prevents from groundless elimination. Hence, in

the context of preferred consistent query answers it is natural to restrict our attention to

families of optimal repairs which satisfy the essential properties \( \mathcal{P} 1 \) and \( \mathcal{P} 2 \).
3.5 Common optimal repairs

Now, we investigate the question whether there are repairs common for any family of optimal repairs that satisfies the properties $P_1$ and $P_2$, i.e. for a given instance $r$, a given set of functional dependencies, and a given priority $\succ$, is there a repair $r'$ which is in $X-\text{Rep}_F(r)$ for any family $X-\text{Rep}$ of optimal repairs satisfying $P_1$ and $P_2$. The answer is negative for families of semi-globally (and thus also locally) optimal repairs. For instance we can construct two families of semi-globally optimal repairs that define the same set of preferred repairs as $S-\text{Rep}$ except that for the setting in Example 9 one returns only $r_1$ while the other only $r_2$. Surprisingly, the situation is different for families of globally optimal repairs.

**Theorem 1.** For every instance $r$, every set of functional dependencies $F$, and any every priority $\succ$, there exists a repair $r'$ such that $r' \in X-\text{Rep}_F(r)$ for any family $X-\text{Rep}$ of globally optimal repairs that satisfies $P_1$ and $P_2$.

We define a new family of $C-\text{Rep}$ which selects only common repairs of all families of globally optimal repairs satisfying the essential properties $P_1$ and $P_2$. $C-\text{Rep}$ is another family of preferred repairs that satisfies all properties.

**Proposition 6.** $C-\text{Rep}$ satisfies properties $P_1$ and $P_4$ and $C-\text{Rep} \subseteq G-\text{Rep}$

Interestingly the family of common repairs has an alternative procedural characterization.

**Proposition 7.** For a given instance $r$, a given set of functional dependencies $F$, and a given priority $\prec$, the set $C-\text{Rep}_F(r)$ consists of all results of Algorithm 1 for any sequence of choices in Step 3.

We also note that under some conditions, the properties $P_1$ and $P_2$ specify exactly one family of globally optimal repairs.

**Theorem 2.** $C-\text{Rep}$ and $G-\text{Rep}$ coincide for priorities that cannot be extended to a cyclic orientation of the conflict graph.

4 Computational properties

In this section we study the computational implications of using priorities to handle inconsistent databases. Because of space restriction we skip the proofs (most of them can be found in or easily based on reductions presented in [8]).

4.1 Data complexity

In our paper we use the notion of data complexity [22] which captures the complexity of a problem as a function of the number of tuples in the database. The input consists of the relation instance and the priority relation, while the database schema, the integrity constraints, and the query are assumed to be fixed. For a family $X-\text{Rep}$ of preferred repairs we study two fundamental computational problems:
(i) \textit{$X$-repair checking} – determining if a database is a preferred repair of a given database i.e., the complexity of the following set

$$\mathcal{B}_F^X = \{(r, r') : r' \in X\text{-Rep}_F(r)\}.$$ 

(ii) \textit{$X$-consistent query answers} – checking if \texttt{true} is an answer to a given query in every preferred repair i.e., the complexity of the following set

$$\mathcal{D}_{F,Q}^X = \{(r, r') : \forall r' \in X\text{-Rep}_F(r). r' \models Q\}.$$ 

4.2 Complexity results

First we state that computing preferred consistent query answer with any family of semi-globally (and thus also globally) optimal repairs that satisfies $P_1$ and $P_2$ leads to intractability.

**Theorem 3.** For any family $X$-Rep of semi-globally optimal repairs that satisfies $P_1$ and $P_2$, there exists a set of two functional dependencies $F$ and a quantifier-free ground query $Q$ (consisting of one atom) to which computing the $X$-consistent answer is co-NP-hard.

It’s an open question whether a similar statement holds for families of locally optimal repairs. We note that computing preferred consistent query answers is co-NP-hard if we consider a slightly restricted locally optimal repairs: locally optimal repairs for which there doesn’t exist a pair of tuples $x_1, x_2$ which can be replaced with a tuple $y$ such that $y \succ x_1$ and $y \succ x_2$ and the resulting set of tuples is consistent. Therefore we state the following conjecture.

**Conjecture 1.** For any family $X$-Rep of preferred repairs satisfying $P_1$, $P_2$, and global local optimality computing $X$-consistent answers is co-NP-hard.

Another argument for this conjecture is the intractability of computing $L$-consistent query answers.

**Theorem 4.** The $L$-repair checking is in PTIME. There exists a set of two functional dependencies and a quantifier-free query (consisting of one atom only) for which computing $L$-consistent answers co-NP-complete.

To find if a repair $r'$ is semi-globally optimal we seek a tuple $yr \setminus r'$ whose all neighbors in $r'$ are dominated by $y$. Such a tuple exists if and only if $r'$ is not semi-globally optimal. The tractability of $S$-checking implies that computing $S$-consistent answers is in co-NP: the nondeterministic machine uses a polynomial in the size of $r$ number of nondeterministic steps to construct a repair $r'$, checks if $r'$ is semi-globally optimal; the machine finds the answer to the query in $r'$ (if $r'$ is not semi-optimal then the machine halts with the answer ‘yes’). With Theorem 3 we obtain:

**Corollary 1.** The $S$-repair checking is in PTIME and computing $S$-consistent answers is co-NP-complete.
Checking if a repair is globally optimal requires, however, an essential use of nondeterminism. This also promotes computing preferred consistent query answers to a higher level of the polynomial hierarchy.

**Theorem 5.** There exists a set of five functional dependencies for which the $G$-repair checking is co-NP-complete. There exists a set of four functional dependencies and a quantifier-free query (consisting of one atom only) for which computing $G$-consistent answers is $\Pi^p_2$-complete.

The procedural nature of common repairs makes it possible to check if a repair $r'$ belongs to $\mathcal{C}$-$\text{Rep}_F^<(r)$ with a simulation of Algorithm 1 with the choices in Step 3 restricted to $\omega_>(r) \cap r'$. Naturally this process can be performed in polynomial time. Again using Theorem 3 we get:

**Corollary 2.** The $\mathcal{C}$-repair checking is in PTIME and computing $\mathcal{C}$-consistent answers is co-NP-complete.

## 5 Related work

We limit our discussion to work on using priorities to maintain consistency and facilitate resolution of conflicts.

The first to notice the importance of priorities in information systems is [9]. The authors study there the problem of updates of databases containing propositional sentences. The priority is expressed by storing a natural number with each clause. If during an update (inserting or deleting a sentence) the inconsistency arises, then the priorities are used in a fashion similar to $G$-repairs to select minimally different repairs. We note, however, that the chosen representation of priorities imposes a significant restriction on the class of considered priorities. In particular it assumes transitivity of the priority on conflicting facts i.e. if facts $a$, $b$, and $c$ are pair-wise conflicting and $a$ has a higher priority than $b$ and $b$ has a higher priority than $c$, then the priority of $a$ is higher than $c$. This assumption cannot be always fulfilled in the context of inconsistent databases. For example the conflicts between $a$ and $b$, and between $b$ and $c$ may be caused by violation of one integrity constraints while the conflict between $a$ and $c$ is introduced by a different constraint. While the user may supply us with a rule assigning priorities to conflicts created by the first integrity constraint, the user may not wish to put any priorities on any conflicts created by the other constraint.

A similar representation of priorities used to resolve inconsistency in first-order theories is studied in [4], where the inconsistent set of clauses is stratified (again the lowest strata has the highest priority). Then preferred maximal consistent subtheories are constructed in a manner analogous to $C$-repairs. Furthermore, this approach is generalized to priorities being a partial orders, by considering all extensions to weak orders. Again, however, this approach assumes transitivity of priority on conflicts, which as we explained previously may be considered a significant restriction.

In [19] priorities are studied to facilitate the process of belief revision. A belief state is represented as an ordered list of propositional formulae and the revision operation simply adds the given sentence at the end of the given belief state. This representation
of belief state allows to keep track of revision history, which is later used to impose a preference order on the possible interpretations of the belief state. Only maximally preferred interpretations are used when defining the entailment relation.

In the context of logic programs, priorities among rules can be used to handle inconsistent logic programs (where rules imply contradictory facts). More preferred rules are satisfied, possibly at the cost of violating less important ones. In a manner analogous to \( \ll \), [21] lifts a total order on rules to a preference on (extended) answers sets. When computing answers only maximally preferred answers sets are considered.

[20] investigate disjunctive logic programs with priorities on facts. A transitive and reflexive closure of user supplied priorities on facts is used to define a relation of preference on models of the program. The definition of preference on models of the disjunctive program is essentially different from the characterization of globally optimal repairs in Proposition 5. The answer to a program in the extended framework consists of all maximally preferred answer sets. The main shortcoming of using this framework is its computational infeasibility (which is specific to decision problems involving general disjunctive programs): computing answers to ground queries to disjunctive prioritized logic programs under cautious (brave) semantics is \( \Pi_3 \)-complete (resp. \( \Sigma_3 \)-complete).

A simpler approach to the problem of inconsistent logic programs is presented in [14]. There conflicting facts are removed from the model unless the priority specifies how to resolve the conflict. Because only programs without disjunction are considered, this approach always returns exactly one model of the input program. Constructing preferred repairs in a corresponding fashion (by removing all conflicts unless the priority indicates a resolution) would similarly return exactly one database instance (fulfillment of \( \mathcal{P}1 \) and \( \mathcal{P}4 \)). However, if the priority does not specify how to resolve every conflict, the returned instance is not a maximal set of tuples and therefore it is not a repair. Such an approach leads to a loss of (disjunctive) information and do not satisfy \( \mathcal{P}2 \) and \( \mathcal{P}3 \).

[10] proposes a framework of conditioned active integrity constraints, which allows the user to specify the way some of the conflicts created with the constraint can be resolved. This framework satisfies properties \( \mathcal{P}1 \) and \( \mathcal{P}3 \) and doesn’t satisfy \( \mathcal{P}2 \) and \( \mathcal{P}4 \). [10] also describes how to translate conditioned active integrity constraints into a prioritized logic program [20], whose preferred models correspond to maximally preferred repairs. We note that the framework of prioritized logic programming is computationally more powerful (computing answers under the brave semantics is \( \Sigma_3 \)-complete) than required by the problem of finding if an atom is present in any repair (\( \Sigma_2 \)-complete). It is yet to be seen if less powerful programming environment (like general disjunctive logic programs) can be used to compute preferred answers.

[17] uses ranking functions on tuples to resolve conflicts by taking only the tuple with highest rank and removing others. This approach constructs a unique repair under the assumption that no two different tuples are of equal rank (satisfaction of \( \mathcal{P}4 \)). If this assumption is not satisfied and the tuples contain numeric values, a new value, called the fusion, can be calculated from the conflicting tuples (then, however, the constructed instance is not a repair in the sense of Definition 1 which means a possible loss of information).

A different approach based on ranking is studied in [13]. The authors consider polynomial functions that are used to rank repairs. When computing preferred consistent
query answers, only repairs with the highest rank are considered. The property $\mathcal{P}3$ is trivially satisfied, but because this form of preference information does not have natural notions of extensions and maximality, it is hard to discuss postulates $\mathcal{P}2$ and $\mathcal{P}4$. Also, the preference among repairs in this method is not based on the way in which the conflicts are resolved.

An approach where the user has a certain degree of control over the way the conflicts are resolved is presented in [12]. Using repair constraints the user can restrict considered repairs to those where tuples from one relation have been removed only if similar tuples have been removed from some other relation. This approach satisfies $\mathcal{P}2$ but not $\mathcal{P}1$. A method of weakening the repair constraints is propose to get $\mathcal{P}1$, however this comes at the price of losing $\mathcal{P}2$.

6 Conclusions and future work

In this paper we proposed a general framework of preferred repairs and preferred consistent query answer. We also proposed a set of desired properties a family of preferred repairs should satisfy. We presented 4 families of preferred repairs: $L$-Rep, $S$-Rep, $G$-Rep, and $C$-Rep. Figure 5 summarizes the computational complexity results; its first row is taken from [6].

<table>
<thead>
<tr>
<th>Repair Check</th>
<th>Consistent Answers to ${\forall, \exists}$-free queries</th>
<th>Possible Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Rep$</td>
<td>PTIME</td>
<td>PTIME</td>
</tr>
<tr>
<td>$L$-Rep</td>
<td>PTIME</td>
<td>co-NP-complete</td>
</tr>
<tr>
<td>$S$-Rep</td>
<td>PTIME</td>
<td>co-NP-complete</td>
</tr>
<tr>
<td>$G$-Rep</td>
<td>co-NP-complete</td>
<td>$\Pi_2^p$-complete</td>
</tr>
<tr>
<td>$C$-Rep</td>
<td>PTIME</td>
<td>co-NP-complete</td>
</tr>
</tbody>
</table>

Fig. 5. Summary of complexity results.

We envision several directions for further work. Along the lines of [2], the computational complexity results could be further studied, by assuming the conformance of functional dependencies with BCNF.

Extending our approach to cyclic priorities is an interesting and challenging issue. Including priorities in similar frameworks [12] of preferences leads to loosing the monotonicity. A modified, conditional, version of monotonicity may be necessary to capture non-trivial families of repairs.

The last is a generalization of our framework to a broader class of constraints. Conflict graphs can be generalized to hypergraphs [6], which allow to handle broader class of denial constraints. Then, more than two tuples can be involved in a single conflict and the current notion of priority does not have a clear meaning.
References


Taming Data Explosion in Probabilistic Information Integration

Ander de Keijzer, Maurice van Keulen, and Yiping Li

Faculty of EEMCS, University of Twente
POBox 217, 7500AE Enschede, The Netherlands
{a.dekeijzer,m.vankeulen,liy}@ewi.utwente.nl

Abstract. Data integration has been a challenging problem for decades. In autonomous data integration, i.e., without a user to solve semantic uncertainty and conflicts between data sources, it even becomes a serious bottleneck. A probabilistic approach seems promising as it does not require extensive semantic annotations nor user interaction at integration time. It simply teaches the application how to generically cope with uncertainty. Unfortunately, without any world knowledge, uncertainty abounds as almost everything becomes (theoretically) possible and maintaining all possibilities produces huge volumes of data. In this paper, we claim that simple and generic knowledge rules are sufficient to drastically reduce uncertainty, hence tame data explosion to a manageable size.

1 Introduction

Information integration largely remains a labor-intensive manual task. At best, tools assist users in the integration with suggestions of matching data items and attributes, or with performing schema and data conversions based on given rules. The need for human interaction is illustrated by the data integration challenges given by [Lev99]: (1) overlapping and contradictory data, (2) semantic mismatches among sources, and (3) different naming conventions for data values. These challenges require a human’s world knowledge to make concrete decisions. Only exact decisions can unambiguously determine the resulting data items. Even AI techniques cannot make such decisions with certainty.

Our work focuses on autonomous information integration. In applications like ambient intelligence, where devices have their own databases and network connectivity is ad hoc, devices need to exchange and integrate information whenever the opportunity arises and without human interaction. Hence, we approach information integration differently: any decision that needs world knowledge, is not resolved, but all possible outcomes are stored with an associated probability.

Unfortunately, without any kind of world knowledge, huge information sources would be produced in this way. This is due to the fact that many things, however remotely possible, are indeed in principle possible. In [KKA05], we calculated that for two information sources with each five data items, there are in theory 1546 possibilities how these may combine. In this paper, we show that this data explosion can be greatly reduced by using simple and generic knowledge rules.
The paper is organized as follows. First, we position our work among related research and summarize our probabilistic XML integration approach. Section 4 subsequently examines a movie information integration scenario. Section 5 introduces simple knowledge rules and attempts to quantify their effect.

2 Related Work

For a survey on information integration, we refer to [DH05]. We distinguish between schema and data integration and focus on the latter. We deal with the aforementioned data integration challenges by explicitly handling the inherent uncertainties using a probabilistic database approach.Suciu’s SIGMOD’05 tutorial comes with an extensive bibliography on the topic of probabilistic data management [SD05]. Originally, work concentrated on relational databases, but in [KKA05] we argue that XML expresses uncertainty in a more natural way. Other probabilistic XML databases are, for example, PXML [HGS03] and ProTDB [NJ02]. Many results from the logic programming and artificial intelligence communities carry over to our probabilistic XML approach.

Schema matching techniques [RB01] can often be adapted and applied to probabilistic databases. For example, duplicate detection, matching and classification techniques can be used to find and assign probabilities to different representations of the same real-world object (rwo).

Finally, an important source of schema and data integration techniques can be drawn from the Semantic Web community. Approaches mostly attempt to sufficiently annotate data with meaning and world knowledge. We approach data integration from the other end: our approach is independent of any world knowledge, but adding some can be used to restrict uncertainty. We believe this to be a more practical approach than to always require enough annotation to take away all uncertainty. Moreover, in this paper we claim that only simple and generic world knowledge statements suffice.

3 Information Integration using Probabilistic XML

In an ordinary XML document, all information is certain. When XML information sources contain data on the same rwo’s, conflicts may occur. Consider, e.g., two address books: one claims that a person’s name is ‘John’ while the other claims it is ‘Jon’. Therefore, after data integration, there may exist more than one possibility for a certain text node, or in general, for entire subtrees. We model this uncertainty in a probabilistic XML tree with three kinds of nodes: (1) probability nodes (∨), (2) possibility nodes (◦), which have an associated probability, and (3) ordinary XML nodes (•). The children of a probability node enumerate all possibilities. Figure 1 shows a probabilistic XML tree illustrating uncertainty about the name of a person.

![Example probabilistic XML tree](image)
A probabilistic XML tree can be seen as a device’s knowledge about the ‘real world’. The probabilistic XML tree of Figure 1 says that in the real world, there exists with certainty one person with telephone number “1111” and named either “John” or “Jon”. A possible database instantiation is called possible world. The answer to a query on a probabilistic XML tree can be determined by executing the query on each possible world separately.

As an example of data integration, consider two address books containing the addresses of a “John1” and “Rita1”, and a “Jon2” and “Rita2”, respectively. Without world knowledge, even “John1” may refer to the same row as “Rita2” resulting in an integration result with 7 possible worlds (see Table 1).

In [KKA05], we give a formalization of notions like probabilistic XML tree, probabilistic information integration, and related properties.

### 4 Movie database scenario

We investigate data explosion in a scenario in which we integrate four data sources on the web containing movie information (see Table 2). We show that simple and generic world knowledge statements can greatly reduce the number of possibilities without negative effects on querying. More details can be found in [KKL06].

The main cause for data explosion is the semantic equality problem: How to decide whether or not two data items refer to the same row? Any movie element, however remotely possible, may in theory be semantically equal to any other movie element from another source. The existence of keys or key-like attributes can almost completely avoid this problem. For movies we found two candidates: Semantic equality of nearly all movies can be established with the IMDb-number or the combination of title and year. Note that probabilistic integration only calls for safely ruling out possibilities, hence does not require a perfect key. For example, the movie title alone would reduce the number of possible matches for IMDb’s “King Kong/2005” with AMG from 290,000 to 3.

### Table 1. Possible worlds

<table>
<thead>
<tr>
<th>Person 1</th>
<th>Person 2</th>
<th>Person 3</th>
<th>Person 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>John1</td>
<td>Rita1</td>
<td>Jon2</td>
<td>Rita2</td>
</tr>
<tr>
<td>John1=Jon2</td>
<td>Rita1</td>
<td>Jon2</td>
<td></td>
</tr>
<tr>
<td>Rita1=Jon2</td>
<td>John1</td>
<td>Rita2</td>
<td></td>
</tr>
<tr>
<td>Rita1</td>
<td>Rita2</td>
<td>John1</td>
<td>Jon2</td>
</tr>
<tr>
<td>John1=Jon2</td>
<td>Rita2</td>
<td>Rita1</td>
<td></td>
</tr>
<tr>
<td>John1=Jon2</td>
<td>Rita1</td>
<td>Rita2</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2. Some movie sources

<table>
<thead>
<tr>
<th>Source</th>
<th>#movies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Movie Database (<a href="http://www.imdb.com">http://www.imdb.com</a>)</td>
<td>470,000</td>
</tr>
<tr>
<td>All Movie Guide (<a href="http://www.allmovie.com">http://www.allmovie.com</a>)</td>
<td>290,000</td>
</tr>
<tr>
<td>Yahoo! movies (<a href="http://movies.yahoo.com">http://movies.yahoo.com</a>)</td>
<td>1,500</td>
</tr>
<tr>
<td>Simply Scripts (<a href="http://www.simplyscripts.com">http://www.simplyscripts.com</a>)</td>
<td>unknown</td>
</tr>
</tbody>
</table>

### Table 3. Comparison of information on the 2005 movie “King Kong”.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title/Year</td>
<td>Exactly equal in all three sources.</td>
</tr>
<tr>
<td>Genre</td>
<td>IMDb: Action, Adventure, Drama. Fantasy, Sci-Fi, Thriller. AMG: Adventure, Monster Film, Period Film. Yahoo: Action/Adventure, Romance, Thriller, Remake.</td>
</tr>
<tr>
<td>Cast</td>
<td>IMDb: 15 people, AMG: 11 (all also in IMDb and Yahoo), and Yahoo: 13 (extra 2 are different from the 4 extra of IMDb). 3 differences in spelling.</td>
</tr>
<tr>
<td>Plot summary</td>
<td>All three sources have a different description or plot summary.</td>
</tr>
</tbody>
</table>
Other causes for data explosion are differences in attribute existence and values. However, these uncertainties are local for an attribute and storage overhead is expected to be small using the compact representation of [KKA05]. Querying the resulting integrated source is not expected to suffer significantly from the incurred uncertainty. Items can still be found, some items may only have a reduced probability. Table 3 illustrates these conclusions for one movie. For genre, cast and transcript, we use a generic rule for lists of text nodes: no semantic equality if two strings sufficiently mismatch. For example, integrating IMDb’s and Yahoo’s genre attributes results in only three uncertainties: ‘Action/Adventure’ is the same as ‘Action’ or ‘Adventure’, or is an entirely different genre. For location and plot summary, we use a generic cardinality rule: the schema requires only one value, hence a different value between sources means another possibility. Observe that a query asking for movies filmed in New Zealand containing a predicate like location='New Zealand', will find the movie “King Kong” in the integrated source.

5 Simple Knowledge Rules

The framework of [KKA05] is independent of any world knowledge for integration of information sources. In theory, any element from one information source may refer to the same row as any element of another. Hence theoretically, the number of possibilities in the resulting information source is huge. To more concretely quantify the effects of simple knowledge rules on the number of possibilities, we conducted experiments on the two example data sources of [KKA05] which contained four and two addresses, respectively (see Figure 2 for the DTD of both sources).

We defined several simple knowledge rules that are based on numbers of attributes being equal between elements and on key-like attributes. Some of the knowledge rules and resulting number of possible worlds can be found in Table 4. The simplest of the knowledge rules, the single element rule, reduced the number of possible worlds from 1546 to 39 (±97.5%). The actual knowledge introduced is very safe and minimal: if two data items do not agree on any attribute, we decide that they do not refer to the same real-world object. Further reductions to 15 or even 3 possible worlds can be obtained.

Table 4. Knowledge rules and resulting number of possible worlds (#pw)

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule</th>
<th>#pw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignorance</td>
<td>No world knowledge, i.e., any two elements may refer to the same row</td>
<td>1546</td>
</tr>
<tr>
<td>Single element</td>
<td>Elements do not refer to the same row, if none of the children have the same value</td>
<td>39</td>
</tr>
<tr>
<td>50%</td>
<td>Elements do not refer to the same row, if less than 50% of the children have the same value</td>
<td>15</td>
</tr>
<tr>
<td>Firstname</td>
<td>The firstname attribute is considered a key, i.e., elements do not refer to the same row, if the firstnames disagree</td>
<td>15</td>
</tr>
<tr>
<td>Lastname</td>
<td>Analogously for lastname</td>
<td>3</td>
</tr>
<tr>
<td>Combination 1</td>
<td>50% and firstname rule</td>
<td>15</td>
</tr>
<tr>
<td>Combination 2</td>
<td>50% and lastname rule</td>
<td>3</td>
</tr>
<tr>
<td>Combination 3</td>
<td>firstname and lastname rule</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 2. DTD of example sources
We should, however, avoid adding world knowledge that does not hold in general. For example, if document 1 would have had the data item ‘John Kingship / phone=4030 / room=3035’, it is actually very likely that this data item does not refer to the same row as ‘Allen Kingship / phone=2020 / room=3035’. The 50% rule is in this case not a good knowledge rule, because it rules out possibilities that are likely to be true. Good knowledge rules for probabilistic integration are safe rules that have little or no false positives.

6 Conclusion and future work

In this paper we have shown that data explosion in probabilistic information integration can be reduced drastically by introducing safe, simple and generic knowledge rules. In the movie database scenario, we looked at some real-life data to be able to investigate the uncertainty occurring in practical information integration. We showed that although much conflicting information can be found, there is enough solid ground. It is expected that the remaining uncertainty need not be resolved to be able to effectively answer the usual queries.

Although probabilistic information integration can function without user interaction at integration time, user interaction may still be beneficial. A user could indicate that certain possibilities are nonsense. In such a case, those possibilities can be eliminated from the source. As future research, we will investigate if user statements about a query result can be used to reduce uncertainty.

References


Model theoretic and fixpoint semantics for preference queries over imperfect data

Peter Vojtáš
Charles University and Czech Academy of Science, Prague
Peter.Vojtas@mff.cuni.cz

Abstract. We present an overview of our results on model theoretic and fixpoint semantics for a relational algebra using a model of many valued Datalog with similarity. Using our previous results on equivalence of our model and certain variant of generalized annotated programs, we base our querying on fuzzy aggregation operators (also called annotation terms, combining functions, utility functions). Using of fuzzy aggregation operators (distinct from database aggregations) enables us to reduce tuning of various linguistic variables. In practice we can learn fuzzy aggregator operators by an ILP procedure for every user profile. Our approach enables also integration of data from different sources via aggregation and similarity. Extending domains we discuss difference between fuzzy elements and fuzzy subsets. We also discuss an alternative, when all extensional data are stored crisp and fuzziness is in rules interpreting data, context and in user query.

Keywords: fuzzy Datalog, preference querying, correct and complete semantics,

Introduction.
Relational data model of E. F. Codd [3] was closely related to logic of predicate calculus. Connections of model theoretic semantics and proof theoretic (possibly also fixpoint) semantic was considered as a standard part of the development of a formal data model (see e.g. J. D. Ullman [15]).

Necessity to extend the expressivity of data modeling languages to include vagueness, preference, uncertainty and different forms of imperfection was stressed by several authors (see [2], [4], [5], [6], [11], [13], [14], just to mention a few).

In this work we would like to present a model theoretic and fixpoint semantics for a relational algebra introduced in [12] and discuss several aspects of the system.

Our motivation is from the work on a system developing methods and tools for acquiring, managing, mining and presenting information in a heterogeneous environment [10]. First pilot application we work on is in the domain of labor market. Here we will apply our techniques of preference queries. Job offer can mention place, required skills, salary and age “young”. Job seeking can be interested in distance “close”, salary “high” and specifying age “about 25”. The problem is to assign a degree of matching between job offer and the user in a way, he gets best answers first. Moreover our system incorporates combining overall score from score for particular attributes (see e.g. [4]). We model combining
function by a special sort of logical connectives - fuzzy aggregation operators (please do not confuse them with database aggregations) - which can range from conjunctions to disjunctions, but most typically are somewhere “in the middle”, expressing fulfillment of most of requirements wrt. some weighting of importance of attributes.

Preference, better answer is modeled using many valued logic - the bigger the truth value the better the answer.

Fuzzy Datalog.

We follow the model of fuzzy logic programming developed in [16] and [8] and implement usual restrictions which make them Datalog programs. Our language is typed $[0, 1]$-valued predicate logic. The only connectives are implications and aggregations $\&$. Aggregations have truth functions fuzzy aggregation operators $\&'$ which are monotone in all variables and $\&'(0, \ldots, 0) = 0$ and $\&'(1, \ldots, 1) = 1$. Aggregations cover all sorts of fuzzy conjunctions and disjunctions. We have no negation here.

Declarative semantics is based on the notion of correct answer. Assume $P$ is a fuzzy positive Datalog program, $Q$ is an atom of our language and $\theta$ is a substitution and $x \in [0, 1]$ is a real number. We say that $\theta, x$ is a correct answer to query $Q$ wrt the program $P$ if for all models $\mathcal{M}$ of the program $P$ the model assigns to $Q\theta$ a truth value $\mathcal{M}(Q\theta) \geq x$.

Procedural semantics of fuzzy Datalog we are using here was described in [8] and is based on the many valued modus ponens and residual conjunctors $C_i$ to rule implicators $\rightarrow_i$

$$
(B.b), (H \leftarrow_i B.r) \quad \frac{}{(H.C_i(b,r))}.
$$

The Datalog production operator is defined as follows: Assume $P$ is a fuzzy definite logic program and $F = \mathcal{B}^{[0,1]}$ be the complete lattice of all fuzzy Herbrand interpretations (ordered coordinate wise). Then for $\delta \in F$ and a ground atom $H$ the Datalog production operator is defined as follows:

\[ T_P(\delta)(H) = \max\{\sup\{\delta^*(\&(B_1, \ldots, B_m)) : (H \leftarrow L \&(B_1, \ldots, B_m)) \text{ is a ground instance of a rule in the program } P\}, \sup\{c : (H.c) \text{ is a ground instance of a fact in the program } P\}\}. \]

The $T_P$ operator is continuous provided all truth functions of fuzzy aggregations in body of rules are left continuous (in the sense of functions of real numbers) in all variables.

Similarly as in the classical case fuzzy models are characterized by following:

**Theorem.** Assume $P$ is a definite fuzzy logic program. A Herbrand structure $\delta$ is a model of $P$ iff $T_P(\delta) \leq \delta$ (hence the minimal fixpoint of $T_P$ is a minimal Herbrand model of $P$).

For corresponding fuzzy logic programming we have following

**Theorem.** Assume our language contains only left continuous annotations, $P$ is a fuzzy definite logic program, $\theta$ is a substitution and $x \in [0, 1]$. Then

*(soundness)* if $\theta, x$ is a computed answer to $Q$ with respect to $P$, then $\theta, x$ is a correct answer.
if \( \theta, x \) is a correct answer to \( \neg Q \) with respect to \( P \), then for every \( \epsilon > 0 \) there is a computed answer \( (y, \vartheta) \) to \( \neg Q \) with respect to \( P \) such that \( x - \epsilon < y \) and \( \vartheta = \theta \gamma \) (for some \( \gamma \)).

Our system captures arbitrary fuzzy similarity, not only max-min (see [9]) simply extending arbitrary logic program by axioms of similarity (which have form of rules).

**Fuzzy relational algebra.**

Fuzzy relational algebra was developed in [12]. The main point we would like to stress here is our join parameterized by a fuzzy aggregation operator. It is defined wrt crisp equality and a fuzzy aggregation operator which tells us how to calculate the truth value degree of a tuple in the join. Assume we have relations \( R_1, \ldots, R_n \) which evaluate predicates \( r_1, \ldots, r_n \). Moreover assume that first \( k \)-attributes in each \( R_i \) are the same and \( (b_1, \ldots, b_k, b_{k+1}, \ldots, b_{m_1}, \beta^i) \in R_i \) then \( (b_1, \ldots, b_k, b_{k+1}, b_{m_1}, \ldots, b_{m_n}, \beta^i) \in \mathcal{R}_\alpha (R_1, \ldots, R_n) \), that is the truth value attributes in our join do not behave as in classical join, they disappear, forwarding the respective truth values to the new aggregated truth value.

Besides this, our algebra contains also selection \( \sigma_{\text{TruthV} \geq t}(r) \), similarity closure \( \mathcal{S} \mathcal{Y} \mathcal{M}_{\alpha \cup \mathcal{M}}(R) \), classical projection \( \Pi_{X_1, \ldots, X_h}(R) \) and max-union.

**Theorem.** Every query over the fuzzy knowledge base represented by a fuzzy positive Datalog program (possibly with recursion, without negation) extended by rules describing the equational theory of predicate calculus and properties of fuzzy similarities can be arbitrarily exactly evaluated by iterating described operations of fuzzy relational algebra.

**Learning the fuzzy aggregation operator.**

An advantage of our approach is that we can learn the aggregation function from user’s classification by a fuzzy ILP system developed in [17]. We use equivalence ([8]) of our programs to GAP ([7]) and inducing annotation terms.

We assume a set of job offers \( J \) with database attributes ([10]). Assume a user classifies (a representative sample of) job offers by \( C: J \rightarrow [0, 1] \). We translate this to multiple classical ILP task using \( \alpha \)-cuts and special monotonicity axioms

\[
\begin{align*}
\mathcal{C} & \xrightarrow{\alpha\_\text{cut}} \mathcal{C}_\alpha \\
\mathcal{C}_\alpha \cup \mathcal{M} & \xrightarrow{\text{ILP}} \mathcal{G}_\alpha
\end{align*}
\]

Background knowledge has to be extended by monotonicity axioms \( \mathcal{M} \). For each predicate touched by user preference and fuzziness, we add e.g.

\( \text{near}(h, x) \leftarrow \text{near}(h, y) \) for all \( x \leq y \).

Then classical ILP learning takes into account "better" examples and does not violate ordering of classification.
Some problems and future research directions.

Fuzzy tuples versus fuzzy attributes. Some fuzzy data approaches differ in the way assigning fuzzy truth value to the whole tuple or to each attribute value. In our approach we assign fuzzy truth value to the whole tuple. In our opinion it covers the fuzzy attribute case in the following sense. Many $n+1$-ary relations $p(A_0, \ldots, A_n)$ from practical applications can be decomposed into $n$-many binary relations $p_i(A_0, A_i)$. Here $A_0$ plays the role of a resource URI and $A_i$ is the property, which in instances corresponds to property value (this corresponds to modeling sufficiency of the RDF data model used in the semantic web, [18]). Moreover in our applications we do not store any fuzzy data. Even values like “close” and “young” are stored as terms (ASCII strings) and only in the time of query evaluation (or query preprocessing) we interpret them and depending on the context and users understanding a fuzzy interpretation is assigned.

Uniform fuzzy small, medium, large and special indexing for efficient search. In many approaches to fuzzy data management we can see various implementations of linguistic variables (like young) and hedges (like very young). In our approach, in each domain with some natural ordering, we use only one universal fuzzy linguistic variable for small, large. Those attain value 1 only at max or min element of the domain. Reason is, that every original rule system can be rewritten using universal fuzzy sets and a new fuzzy aggregation operator. Roughly speaking, the aggregation operator can compensate all differences between users different intentions (unique for users with same profile). Further consideration are devoted to implementation of “medium”. A special indexing structure $B^+_d$ tree is constructed to traverse along the ordering induced by fuzzy sets on domains and to find best (top-k) answers.

Extending domains by fuzzy elements and sets and selection conditions. Another problem occurring in fuzzy data modeling is extensions of domain by fuzzy values. We have made some initial acquaintance with using different models for fuzzy elements of domains (like “about $25$” to be compared with other domain values) and fuzzy subsets for range queries (like “young”). We think, that calculating degree of “about $25 \in$ young” is more appropriate than degree of “about $25 =$ young”. Moreover pure set theoretic semantics of handling those values is not satisfactory for application. In some case we have to use approach form metric spaces (measuring distance between sets), measure theory and other Information Retrieval measures. We mention many possible definitions of fuzzy less (greater) or equal.

Future research directions. In future we plan practical experiments on a system developed at our department. In [1] authors describe a system which implements integration between a heterogeneous set of databases. One of key attribute of their solution is conflict resolution. In future we would like to experiment with some fuzzy similarities learned to fit requirements.

In [10] some initial experiments with real data were done and tests with fuzzy ILP and variants of heuristics arising from Fagin’s threshold algorithm ([4]). We would like to use this to push forward the model of fuzzy data and query model.
Conclusion. In this paper we have reviewed some of our former models giving a model theoretic and fixpoint semantics for fuzzy Datalog programs with similarity and discussed some related issues and future work.

Acknowledgement. Supported by Czech IT projects 1ET100300517, 1ET100300419 and the center of excellence MS0021620838.

References

14. Y. Takahashi. Fuzzy Database Query Languages and Their Relational Completeness Theorem, IEEE Transactions on Knowledge and Data Engineering archive 5 (1993) 122 - 125
18. http://www.w3c.org
Consistency Management Framework for Realtime Disaster Information Systems

NODA, Itsuki
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
2-41-6 Aomi, Koto-ku, Tokyo 135-0064, JAPAN
i.noda@aist.go.jp

Abstract. In this article, I propose a mathematical framework of consistency management that checks validity among data that are used in integrated analysis/simulation for huge disasters, under which multiple and delayed information can be reported with errors to the database continuously in real-time applications.

1 Introduction

Integration of widely-varied related information is the first and the most important step of rescue activities in huge disasters. Databases will play a key role in such situation.

We need to pay attention to error and delay in the reports about disaster damages. Especially, following two cases cause serious inconsistency and non-monotonic changes in the database:

1. Multiple data can be reported for an identical phenomenon. For example in Chuetsu Earthquake occurred Nov. 2004 in Japan, ‘magnitude 5’ and ‘magnitude 7’ were reported alternately as a value of a seismic intensity of a certain aftershock at the same point. Finally, ‘magnitude 7’ was determined as an actual value later, but there was no way to judge which value is a true one.

2. Some data reach to rescue headquarters with large delay. For example in the same earthquake, damage of Yamakoshi village in Niigata were reported to the rescue headquarters two days after the main attack because of the complete black-out of power and telephone lines to this area.

Another aspect of the disaster information is that complicated dependency among multiple phenomena. For example, damages by an earthquake are caused not only by direct attack of the quake but also by secondary and/or side effects like flood and fire. In addition, social confusion like traffic congestion affects the damages and rescue-activities. Therefore, we need take these various factors in account in order to estimate final damage and to make a rescue plan. Actually, we are developing an integrated disaster-and-rescue simulation system [1, 2] in a Japanese national special project for earthquake disaster mitigation in
urban areas (DDT project). In the system, a number of simulators for various phenomena, for example, damages of ground and road, TSUNAMI, fire-ignition, liquefaction, and so on, are used to provide initial data for the multi-agent rescue simulation.

The combination of the inconsistency of the database and the complicated dependency among multiple phenomena in huge disaster brings more serious issues. Suppose that there are phenomena \(X, Y, Z, W\), and in an integrated simulation, \(Y\) and \(Z\) can be estimated from \(X\) value by simulations independently, and \(W\) can be calculated from \(Y\) and \(Z\). Suppose \(X\) is sensed by two sensors \(a\) and \(b\), which report different values \(x_a\) and \(x_b\), respectively. Here, a simulation calculates \(Y\) using \(x_a\) and outputs \(y_a\) as its value. In the same time, another simulation calculates \(Z\) as \(z_b\) based on \(x_b\). In this case, we should avoid calculating value \(W\) by \(y_a\) and \(z_b\), because these values are grounded on different values of phenomenon \(X\).

Similarly, the delay of report causes another issue. Because it is difficult to acquire all data for a certain simulation, we must estimate lacked data by interpolating or other methods to run the simulation. When the estimated data turns out to be different from the real data that are reported with delay, we must ignore results of the simulation and re-calculate using the new real data.

### 2 Formalization

We suppose that a database is a collection of data, each of which indicates a sensed or simulated value about a thing at a certain time. We also suppose that the database is dynamic. In other words, sensors and simulators put new data into the database continuously. \(^1\)

The words thing and data are defined as follows: A thing \(\theta\) is an identifier of an object, feature, or phenomena like 'building X' and 'seismic intensity at a point Z'. An event \(e = \langle \theta, t \rangle\) is a snapshot of a thing \(\theta\) at a time \(t\). A data \(d_{\theta,t,s}\) is a value of event \(\langle \theta, t \rangle\) acquired by a sensor or simulation \(s\). Because a sensed or simulated value is affected by several conditions and noises, multiple data for a certain event can exist. When we indicate a time to store the data into a database, we use the following notation \(d_{\theta,t,s,\tau}\), where \(\tau\) indicates the time when the data is stored into a database.

A database \(DB = \{d_{\theta,t,s,\tau}\}\) is defined as a set of whole data stored in the database. In other words, \(DB_{(0,T)}\) is a set of data that are stored into \(DB\) before \(T\).

We say a session of a simulation to indicate a run of the simulation using certain initial data and settings. Generally, a session is performed as follows:

1. Collect a subset of data \(D_{\text{ground}}\) about some events \(E_{\text{ground}} = \{e_i\}\) from \(DB\).

\(^1\) We assume that data in a database are never updated or deleted. When various values for a certain event are reported, then all values are stored independently in the database.
2. Estimate a dataset $D_{\text{target}}$ about other events $E_{\text{target}} = \{e_i\}$.
3. Store $D_{\text{target}}$ into DB.

Each dataset of input or output of a session is annotated by a label called version in order to indicate how to construct the dataset. A version $v$ is defined as a tuple $(E, T, D, G)$, where $E = \{e_i\}$ is a set of events, $D$ is a dataset under the version, $T$ is a time when DB is accessed to manipulate the dataset, and $G = \{u_i\}$ is a set of versions on which $D$ is directly grounded.

There are four types of operation to generate a new version, sprout, extraction, production, and union. A version is sprouted when a new dataset is collected from DB, where all data in the dataset should be primary one that come from sensors directly. A sprouted version consists of a tuple $(E, T, D, \phi)$, where $E$ is a set of related events, and $D$ is a collected dataset from DB. A version $v$ can be extracted from another version $u = (E_u, T_u, D_u, G_u)$, where the dataset of $v$ $(D_v)$ is formed as a subset of $D_u$. The extracted version $v$ is denoted as a tuple $(E_v, T_v, D_v, \{u\})$, where $D_v \subseteq D_u$ and $E_v = \{e|d_{\theta,t,s} \in D_v, e = \{\theta, t\}\}$. When a session of a simulation outputs a version $v$ as a result using version $u$ as a ground, we call the version $v$ as a production of the version $u$, and denote $u \triangleright v$.

The production $v$ is defined as a tuple $u \triangleright v \Leftrightarrow v = (E_v, T_v, D_v, \{u\})$, where $E_v$ is a set of simulated events, and $D_v$ is the result of the simulation. A union of two versions $u$ and $v$ $(u \oplus v)$ is a version to imply a sum of the two versions. A union is a virtual version that includes empty sets of events and data, so that the unified version is constructed by $u \oplus v = \{\phi, T, \phi, \{u, v\}\}$.

We define a version $u$ is grounded in version $v$ (or denote $u \triangleright u$) iff $v$ is generated by union, production, or extraction from $u$. The grounded relation is recursively extended A relation footnote, denoted as $u \triangleright^* v$, is a recursively extended version of the ground-ness relation.

We define that two versions, $u$ and $v$, are primarily consistent (or denote $u \sim v$) iff $\forall e \in E_u \cap E_v: D_u/e = D_v/e$. In other words, two primarily consistent versions consist of identical dataset for each shared event. Therefore, these two versions can be used for a simulation/analysis in the same time.

We say that two versions, $u$ and $v$, are consistent (or denote $u \sim v$) iff $\forall u', v, v' \triangleright v: u' \sim v'$. In order to guarantee a unified version $u \oplus v$ is dependable, $u$ and $v$ should be consistent.

Finally, we define that a version $v$ is self-consistent iff $v \sim v$. This self-consistency is an important concept, because a self-consistent version is supported only by consistent datasets which can be used in the same time for simulation/analysis. Therefore, we can focus only on keeping and checking self-consistency of newly generated versions in the database.

### 3 Theorems about Keeping Consistency

Using above formalization, we can derive the following theorems.

**Theorem 1** Any sprouted version is self-consistent.
Theorem 2 When a version \( v \) is extracted from a self-consistent version \( u \), \( v \) is self-consistent iff the following condition is satisfied:

\[
\forall v : u \supset v, \quad v \text{ is self-consistent} \iff \forall e \in E_v : D_u/e = D_v/e,
\]

where \( u \) and \( v \) are \( \langle E_u, T_u, D_u, G_u \rangle \) and \( \langle E_v, T_v, D_v, G_v \rangle \), respectively.

Theorem 3 When a version \( u \) is self-consistent, a production \( v \) of the version \( u \) is self-consistent.

\[
\forall u : \text{self-consistent} \rightarrow \\
\forall v : u \triangleright v \rightarrow v \text{ is self-consistent}.
\]

Theorem 4 When two versions, \( v \) and \( u \), are self-consistent and consistent with each other, a union of these versions is self-consistent.

\[
\forall u, v : \text{self-consistent}, u \sim v \rightarrow \\
u \oplus v \text{ is self-consistent}
\]

From these theorems, we can easily construct procedures to keep and check validity of newly generated version which is used as input of simulation / analysis.

4 Discussion

Version control is a common concept in software development. There exist several tools, for example, CVS and MAKE, to handle versions in programs under development.

CVS (Concurrent Versions System) provides facilities to manage dependencies among source codes. Basically, dependencies handled by CVS form a tree structure where nodes and links of the tree are sets of source codes and dependencies between the sets, respectively. On the other hand, to answer the above requirements, we need to handle dependencies that form a lattice structure. Therefore, it is difficult to utilize CVS or similar version control techniques for our purpose.

MAKE, an UNIX command to maintain groups of programs, provides dependency checking mechanism among parts of programs. Dependencies handled by MAKE can form a lattice structure. However, MAKE can not a way to represent inconsistency among program parts. So, it is hard to apply MAKE or similar system to handle inconsistent data of disaster information.

ATMS (Assumption based Truth Maintenance Systems) [3] is also a candidate to handle our requirements. Actually, ATMS provides strong facilities to maintain complicated dependencies among inconsistent data sets. However, ATMS is too heavy for our purpose, because we need to handle huge number of information in a large scale disasters. For example, we may need to handle over million data entries to analyze/simulate disasters of Kawasaki city, which is a
satellite city of Tokyo. This size of problem is too large to utilize the current
ATMS technology. On the other hand, the proposed formalization will provide
more simple and effective checking mechanism [4].

There are several open issues on the formalization as follows:

- How to realize a facility to extract version that is consistent with a certain
version. Such operation will happen when a simulation requires parts of
result of two versions.
- How to formalize statistic operations of results of multiple simulations. One
of purposes of the simulation is to calculate statistic value like averages and
variances based on multiple simulation using different random seeds. Current
formalization can not handle it, because such operations break consistency
among versions.

References

1. Tadokoro, S.: An overview of japan national special project daidaitoku (ddt) for
earthquake disaster mitigation in urban areas. In: Proc. of IEEE International Sym-
posium on Computational Intelligence in Robotics and Automation (CIRA2003)
Workshop on Advanced Robots and Information Systems for Disaster Response.
(2003)
2. Tadokoro, S.: Japan government special project: development of advanced robots
and information systems for disaster response (daidaitoku) — for earthquake dis-
aster mitigation in urban areas —. In: Proc. of IEEE Workshop on Safety Security
4. NODA, I.: Consistency management framework for database used in integrated sim-
ulations. In: Prof. of RoboCup International Symposium 2005, RoboCup Federation
(2005)
Dealing with inconsistencies and incompleteness in database update (position paper)

Giuseppe De Giacomo¹ and Maurizio Lenzerini¹ and Antonella Poggì¹,² and Riccardo Rosati¹

¹ Dipartimento di Informatica e Sistemistica “Antonio Ruberti”
Università di Roma “La Sapienza”
Via Salaria 113, I-00198 Roma, Italy
surname@dis.uniroma1.it
² INRIA Futurs
Parc Club Orsay-University
4 rue Jean Monod, F-91893 Orsay Cedex, France

Several areas of research and various application domains have been concerned in the last years with the problem of dealing with incomplete databases. Data integration as well as the Semantic Web are notable examples. Surprisingly, while many research efforts have been focusing on several interesting issues related to incomplete databases, as query answering, not much investigation have been done concerning updates. In this position paper we aim at highlighting some of the issues we are dealing with in our work on updates over incomplete databases.

**Instance level updates under constraints** Our interest in this area stems mainly from the need to deal with updates in Description Logics based ontologies. Description logics (DLs) are logics for expressing the conceptual knowledge about a domain in terms of classes and associations between them [1]. Such logics are currently considered among the most promising formalisms for representing ontologies by the Semantic Web community [2]. DL-based ontologies are often used for accessing data stored in a data layer by means of query answering. Interestingly, the open world assumption is enforced and incomplete information on the data is assumed in this setting. While there is a whole body of research on query answering in such systems, the research on update is very recent [5, 4]. First results show that classical approaches on logical databases update, such as Winslett’s [7, 8], are often adequate, as long as one takes into account a clear distinction between intensional (conceptual level) information and extensional (instance level) information typical of this setting. Intensional information, expressed by universal assertions in DLs, is considered immutable, while extensional information, expressed in a form of an incomplete database, is subject to updates. From a database point of view, the setting above requires investigating updates on incomplete databases under a wide variety of constraints, such as keys and foreign keys, more general forms of inclusion dependencies, etc. Thus, such research is revamping interest in updates in databases with incomplete information.

**Update expressibility** The problem of update expressibility arises [5] as soon as we consider expressive schema and constraints languages. This problem aims
at deciding whether given a class of incomplete databases $C$ satisfying certain constraints and an incomplete database $I$ belonging to $C$, the result of an update over $I$ is expressible through a new incomplete database belonging to $C$. For many classes $C$, we have that updates are not expressible. One notable such class is the class $K FK$ of incomplete databases that are characterized by a relational schema with keys and foreign keys. Notice that, in contrast, query answering in such class is polynomial (actually LOGSPACE) in data complexity [3].

**Materialized update vs. virtual update** Even if the result of the update is not expressible in a given class $C$ of incomplete databases, we might be interested in answering queries on the database resulting from the update. Indeed, to do so we do not necessarily need to “materialize” the new state of the database, but actually we could reason on the original database base by taking into account the update in a “virtual” way. In a sense, this is analogous to the distinction between projection via regression vs. progression in reasoning about actions well-known in AI [6]. Along this direction, one interesting problem is understanding whether answering queries after updates over incomplete databases belonging to the previously mentioned class $K FK$ remains polynomial.

**Update and inconsistency** Finally, it is worth noticing that updates bring in the general issue of dealing with inconsistency in databases with incomplete information. The standard update semantics addresses the issue of solving inconsistency between the current instance level information (i.e., the data) and what has been asserted by the update, while it does not deal with inconsistencies between the update and intensional level information (i.e., the constraints). It would be interesting to study possible semantics that are tolerant w.r.t. the latter form of inconsistency.

**References**

Querying Incomplete Data: 
Towards Practical Cases 
(position paper) 

Andrea Calì 
Faculty of Computer Science 
Free University of Bozen-Bolzano 
piazza Domenicani 3, I-39100 Bolzano, Italy 
email: ac@andreacali.com 

1 Introduction 

When integrity constraints are expressed on a database schema, it is possible that such constraints are not satisfied by the data represented by the schema. This may happen, for example, in a data warehouse where data are retrieved from different sources, or in a data integration system where the schema is a representation of several independent data sources. In such cases the constraints are not representing some property of the data; instead, they are used to enhance the database schema in order to better represent the domain of interest. 

In case of inconsistencies, the goal is to provide consistent answers, according to a suitable semantics. In this paper we present some results about querying inconsistent databases, analysing decidability and complexity of query answering under different classes of integrity constraints. In particular, data complexity, i.e. complexity w.r.t. the size of the data, is especially important, since the size of the data is usually way larger than that of the schema and constraints. 

We first consider query answering under (subclasses of) the most common database dependencies, namely key and inclusion dependencies. We show how far we can go with expressiveness in this case. 

Then, we draw our attention on conceptual data models. Such models, and in particular the Entity-Relationship (ER) model [10], play a fundamental role in database design. Conceptual schemata used in database design have the necessary expressiveness and flexibility for effectively representing the domain of interest, and are precise enough to allow the implementation on DBMSs. We do believe that, in order to address practical cases, we should consider query answering where the schema is a conceptual model; indeed, most real-world databases are designed by making use of a conceptual schema. 

We consider an extended ER (EER) model, and we represent EER schemata with a special class of key and inclusion dependencies (which is, by the way, what happens in most real-world cases). We show that answering queries over incomplete data in such a setting is decidable and tractable w.r.t. data complexity.
A semantics for incomplete data. We assume the reader is familiar with the relational model [11] and with the language of conjunctive queries (see e.g. [1]). In the following we shall refer to two distinct, fixed and infinite alphabets of constants and fresh constants respectively, and we consider only databases over them. The database constraints of our interest are functional dependencies (FDs), inclusion dependencies (IDs) and key dependencies (KDs) (see e.g. [1]).

Queries on incomplete data. Consider a database \( D \) for a schema \( \mathcal{R} \), on which a set \( \Sigma = \Sigma_K \cup \Sigma_I \) of integrity constraints is expressed, where \( \Sigma_K \) is a set of KDs, and \( \Sigma_I \) is a set of IDs. Suppose that \( D \) satisfies \( \Sigma_K \), written \( D \models \Sigma_K \). In the sound\(^1\) semantics for incomplete data, we consider the data in \( D \) as sound, but not complete with respect to \( \Sigma \); in other words, if \( D \) does not satisfy \( \Sigma_I \), we deduce the existence of more data than those in \( D \) so that \( \Sigma_I \) is satisfied. This forces us to consider multiple databases in order to reason on incomplete information [13, 6].

Formally, given a relational database \( D \) for a schema \( \mathcal{R} \), and a set \( \Sigma \) of integrity constraints, and given a conjunctive query \( Q \) of arity \( n \) over \( \mathcal{R} \), the answers to \( Q \) on \( D \) under \( \Sigma \), denoted \( \text{ans}(Q, D, \Sigma) \), are all the \( n \)-tuples \( t \) such that \( t \in Q(B) \) for all databases \( B \) for \( \mathcal{R} \) such that \( B \models \Sigma \) and \( B \supseteq D \).

2 Answering Queries under KDs and IDs

In this section we show some results about querying incomplete data under key and inclusion dependencies. We recall that we keep the assumption of sound semantics.

In the following, when dealing with the complexity of the query answering problem, we will refer to the decision problem of query answering: given a query \( Q \) and a tuple \( t \) of the same arity, and a database \( D \), determine whether \( t \in Q(D) \).

To solve the query answering problem, we have a powerful notion that is called chase: the chase [12] is a (possibly infinite) database that is obtained by “repairing” the initial database \( D \) according to the IDs and KDs. In particular, there are two chase rules. The ID chase rule adds a tuple to repair the violation of an ID, where unknown values are represented by distinct fresh constants; the FD chase rule repairs violations of FDs found in pairs of tuples by forcing suitable symbols to be equal, where a fresh constant can be turned into a non-fresh one, but not vice-versa. Notice that the FD chase rule may fail, while the ID chase rule may not. An iterative application of the chase rules yelds the chase of a database under a set of constraints, denoted \( \text{chase}_\Sigma(D) \). To check whether a tuple \( t \) is in the answers to \( Q \) over an incomplete database \( D \) under a set \( \Sigma \) of integrity constraints, it is suuffcient to check whether \( t \) is in the answer

\(^1\) Other kinds of semantics are possible, but in this paper we will deal only with the sound one.
to answering queries under the same class of constraints. Then, it is shown that implication of generic FDs and IDs can be reduced to implication of KDs and IDs; in fact, with some technical machinery, it is possible to “emulate” generic FDs by making use of KDs and IDs. This proves the undecidability of query answering under KDs and IDs.

In [12], Johnson and Klug proved (in the context of query containment) that query answering is PSPACE-complete under IDs alone, and also under KDs together with a restricted class of IDs called key-based IDs. Such class of IDs is more general than foreign key dependencies; however, there is still a more expressive class for which query answering is decidable. In [3] and [6] the class of non-key-conflicting IDs (NKCIDs) is introduced. The nice property of this class is the separation theorem, that shows that, if the initial database satisfies the KDs, query answering can be done exactly as if KDs were not present, i.e. taking into account only the NKCIDs; therefore, also query answering under KDs and NKCIDs is PSPACE-complete. The class of KDs and NKCIDs seems to be the most general class for which query answering is decidable, since a slightly more general class of constraints, namely the 1-key-conflicting IDs (1KCIDs), makes the problem of query answering undecidable.

3 Answering queries under conceptual schemata

In this section we present some results related to querying incomplete data, where the schema is expressed in a conceptual model.

The problem was addressed in [5], where a query rewriting techniques allows to retrieve the certain answers in a data integration system where the global schema, representing the whole set of data at the sources, is expressed in a conceptual model derived from Chen’s Entity-Relationship (ER) model [10]. Also, in [7] (in the context of peer-to-peer data integration systems) and [8] the problem is studied in a setting where the schema is expressed in DL-Lite, a simple Description Logics that is capable of capturing several ontology languages and conceptual data models.

Here we present some results that go beyond the ones mentioned before, and try to address query answering in a setting that is intended to capture most conceptual schema that are found in the real world. In particular, we consider a global schema expressed in the Extended Entity-Relationship (EER) model, i.e. a model that model incorporates the basic features of the ER model and OO models, including subset (or is-a) constraints on both entities and relationships, and cardinality constraints. As for cardinality constraints, we allow the specification of functional participation (each instance may participate at most once) and mandatory participation (each instance must participate at least once) to a relationship.

In order to be able to query an EER schema, along the line of [5], we represent the schema in the relational model under KDs and IDs; we obtain a special class of KDs and IDs that we call conceptual dependencies (CDs). It is important to notice that the class of CDs does not fall into the known decidable cases;
this because the is-a relations among relationships, in presence of functional participation constraints, lead to IDs that are not non-key-conflicting.

**Example 1.** Consider the EER schema shown in Figure 1, and depicted in the usual graphical notation for the ER model, where the label [1, 2] in the is-a relation between the two relationships denotes that the components of Manages correspond, in their order, to components 1, 2 (in this order) of Works_in, and the cardinality constraint (0, 1) for Employee denotes that each instance of Employee must participate a minimum of 0 times and a maximum of 1 times to Works_in; the cardinality constraint for the participation of Manager to Manages is analogous. The correspondence between the relational predicates and the EER schema is straightforward; anyway, we refer the interested reader to [4] for the details.

Suppose we have a database with the facts manager\((m)\) and works\(\text{in}(m, d)\). If we construct the chase, we obtain the facts employee\((m)\), manages\((m, \alpha_1)\), works\_in\((m, \alpha_1)\), dept\((\alpha_1)\), where \(\alpha_1\) is a fresh constant. Observe that \(m\) cannot participate more than once to works\_in, so we deduce \(\alpha_1 = d\). We must therefore replace \(\alpha_1\) with \(d\) in the rest of the chase, including the part that has been constructed so far.

In spite of the fact that KDs and IDs do interact in sets of CDs, we are still in luck, since the problem is decidable and polynomial in data complexity, i.e. in the size of the database. The proof shows that, as in [12], it is sufficient to limit the chase to an initial segment to answer the query answering problem. The difficulty here lies in the propagation of the application of the FD chase rule, that in principle may affect the whole part of the chase constructed until a certain step; it is possible to prove [4] that the application of the application of the FD chase rule can affect only up to a certain number of “steps” back in the chase.

The decidability of query answering under CDs can be also be derived from [9], a work that shows query containment for a Description Logic that is able to capture the EER model. However, the data complexity in that case is not shown to be polynomial like in our case. Also [14] addresses the problem of query containment using a formalism for the schema that is more expressive than the one of [4]; however, the problem here is proved to be \(\text{coNP}\)-hard in data complexity.

The approach of [4], besides showing tractability of the problem of answering queries over expressive conceptual schemata, sheds more light on the query answering problem, paving the way for more clever query answering techniques than the naive construction of a portion of the chase of the initial database.
In this paper we did not consider the fact that a single violation of a KD leads to a trivial case for query answering in the (strictly) sound semantics, which makes query answering unpractical; however, the techniques presented here can be integrated with other approaches (see e.g. [2,6]) that deal with KD violations in a smoother way.

Acknowledgments. This work was supported by the EU project TONES (IST-007603). I wish to thank Maurizio Lenzerini, Michael Kifer, Leopoldo Bertossi and Diego Calvanese for fruitful discussions about this topic.

References

A Note on Database Repairing by Value Modification

Jef Wijsen

Université de Mons-Hainaut, Mons, Belgium,
jeff.wijsen@umh.ac.be
WWW home page: http://staff.umh.ac.be/Wijsen.Jef/

1 Motivation

Repairing an inconsistent database means making the database consistent by applying some minimal change. Database repairing gathered much attention since the seminal article by Arenas et al. [1], where repairs are defined in terms of the sets of deleted and inserted tuples. Later on, several ways for repairing by value modification [2-7] have been proposed as an alternative to repairing by tuple deletion/insertion.

The possible benefit of repairing by value modification can be illustrated by the following example, where the relation EMP is subject to the functional dependencies Name $\rightarrow$ \{Birth, Sex, ZIP\} and ZIP $\rightarrow$ City. The latter fd is violated.

<table>
<thead>
<tr>
<th>EMP</th>
<th>Name</th>
<th>Birth</th>
<th>Sex</th>
<th>ZIP</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td></td>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td></td>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Bergen</td>
</tr>
</tbody>
</table>

Under repairing by tuple deletion, we find two repairs $D_1$ and $D_2$:

$D_1$ | Name | Birth | Sex | ZIP  | City   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td></td>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
</tbody>
</table>

$D_2$ | Name | Birth | Sex | ZIP  | City   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Bergen</td>
</tr>
</tbody>
</table>

Note that the repair $D_1$ deletes the entire record about Tim even though all attribute values, except for City, appear to be correct. We could add these data to $D_1$ and infer Tim’s city from other tuples, which gives us $R_1$:

$R_1$ | Name | Birth | Sex | ZIP  | City   |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td></td>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td></td>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
</tbody>
</table>

In moving from EMP to $R_1$, we have modified Tim’s city from Bergen into Mons.

2 Fixes and Ureparirs

We define the essence of a method for repairing by value modification, which was first proposed in [8] and later improved in [7].
<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sex</th>
<th>ZIP</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Y</td>
</tr>
</tbody>
</table>

\[
\text{F}_{1} \quad \text{F}_{2} \quad \text{F}_{3} \quad \text{F}_{4}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sex</th>
<th>ZIP</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Bergen</td>
</tr>
</tbody>
</table>

\[
\text{R}_{1} \quad \text{R}_{2} \quad \text{R}_{3} \quad \text{R}_{4}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth</th>
<th>Sex</th>
<th>ZIP</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>An</td>
<td>1964</td>
<td>F</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Ed</td>
<td>1962</td>
<td>M</td>
<td>7000</td>
<td>Mons</td>
</tr>
<tr>
<td>Tim</td>
<td>1960</td>
<td>M</td>
<td>7000</td>
<td>Bergen</td>
</tr>
</tbody>
</table>

\[
\text{F}_{1} \quad \text{F}_{2} \quad \text{F}_{3} \quad \text{F}_{4}
\]

\[
\theta_{X \rightarrow c}(F_{3}) \quad \theta_{X_{1} \rightarrow c_{1} \cdot X_{2} \rightarrow c_{2}}(F_{4})
\]

**Fig. 1.** Fixes and uprepairs for the relation EMP.

We assume a fixed schema \(\langle A_{1}, A_{2}, \ldots, A_{n} \rangle\), where \(A_{1}, A_{2}, \ldots, A_{n}\) are distinct attributes. A tableau is a finite set of tuples \(\langle s_{1}, s_{2}, \ldots, s_{n} \rangle\) where each \(s_{i}\) is a variable or a constant. A tableau is called linear if no variable occurs more than once in it. A relation is a tableau without variables.

A homomorphism from tableau \(F\) to tableau \(I\) is a substitution \(\theta\) for the variables in \(F\) such that \(\theta(F) \subseteq I\). If such a homomorphism from \(F\) to \(I\) exists, then \(F\) is said to be homomorphic to \(I\). A homomorphism \(\theta\) from \(F\) to \(I\) is called one-one if it does not identify distinct tuples of \(F\); that is, for all \(t_{1}, t_{2} \in F\), \(t_{1} \neq t_{2}\) implies \(\theta(t_{1}) \neq \theta(t_{2})\). If such a one-one homomorphism from \(F\) to \(I\) exists, then \(F\) is said to be one-one homomorphic to \(I\), denoted \(I \sqsupseteq F\).

Given a relation \(I\) and a set \(\Sigma\) of integrity constraints, a fix of \(I\) under \(\Sigma\) is a maximal (under \(\sqsupseteq\)) linear tableau \(F\) that is one-one homomorphic to \(I\) and (not necessarily one-one) homomorphic to some consistent relation; every minimal (under set inclusion) consistent relation \(R\) to which \(F\) is homomorphic, is then called an uprepair.

Figure 1 shows the four fixes for the current example. The fixes \(F_{1}\) and \(F_{2}\) determine the uprepairs \(R_{1}\) and \(R_{2}\) respectively. The fix \(F_{3}\) gives rise to multiple uprepairs, each of which is obtained by replacing \(X\) by some constant \(c\) distinct from 7000. Likewise, uprepairs are obtained from \(F_{4}\) by replacing \(X_{1}\) and \(X_{2}\) by constants \(c_{1}\) and \(c_{2}\) both distinct from 7000.

## 3 Related Construct

We discuss some (dis)similarities with a closely related construct proposed in [6]. Let \(\text{dom}\) be the set of constants. An update of relation \(I\) over schema \(\langle A_{1}, \ldots, A_{n} \rangle\)
is a total function $\delta : I \times \{A_1, \ldots, A_n\} \rightarrow \text{dom}$. An element of $I \times \{A_1, \ldots, A_n\}$ will be called an entry in $I$ (or $I$-entry). Intuitively, if $\delta(t, A_i) = c$ for some $t \in I$, then the update should set the $A_i$-value of $t$ to $c$. The update function $\delta$ is extended to tuples and relations in a natural way:

1. for each $t \in I$, $\delta(t) = (\delta(t, A_1), \ldots, \delta(t, A_n))$; and
2. $\delta(I) = \{\delta(t) \mid t \in I\}$.

Next, we are interested in the $I$-entries that really undergo a change:

$$\text{Modif}(\delta) := \{(t, A_i) \in I \times \{A_1, \ldots, A_n\} \mid \delta(t, A_i) \neq t(A_i)\}.$$  

Since $(t, A_i) \notin \text{Modif}(\delta)$ implies $\delta(t, A_i) = t(A_i)$, the update $\delta$ is fully determined when we know the value of $\delta$ for each $I$-entry of $\text{Modif}(\delta)$. Therefore, when we specify an update $\delta$ hereinafter, we will only specify the values of $\delta$ over $\text{Modif}(\delta)$.

For example, if the three tuples of $\text{EMP}$ are denoted $t_{\text{An}}$, $t_{\text{Ed}}$, and $t_{\text{Tm}}$ respectively, then $R_1 = \delta_1(\text{EMP})$ with $\delta_1 = \{((t_{\text{Tm}}, \text{City}), \text{Mons})\}$, i.e. replace the City-value in $t_{\text{Tm}}$ by Mons (and leave all other entries unchanged). Likewise, for $\delta_2 = \{((t_{\text{An}}, \text{City}), \text{Bergen}), ((t_{\text{Ed}}, \text{City}), \text{Bergen})\}$, we obtain $R_2 = \delta_2(\text{EMP})$.

The update $\delta$ is a set-minimal repair of $I$ under $\Sigma$ if $\delta(I) \models \Sigma$ and for every update $\delta'$, $\text{Modif}(\delta') \subseteq \text{Modif}(\delta)$ implies $\delta'(I) \not\models \Sigma$. Intuitively, a set-minimal repair of $I$ makes $I$ consistent by modifying a minimal (under set inclusion) set of $I$-entries. Under this definition, $\delta_1$ and $\delta_2$ are set-minimal repairs for the running example.

We note incidentally that [6] also introduces the notion of card-minimal repair which minimizes the number of entries modified by the update. Both criterions, minimization with respect to set inclusion and cardinality, have also been used in [3] for repairing census data forms.

The constructs of set-minimal repair and uprepair try to minimize (under set inclusion) the set of modified entries. Given a set-minimal repair $\delta$ of $I$ under $\Sigma$, let $\delta_{\alpha}(I)$ be the tableau obtained from $I$ as follows: whenever $(t, A_i) \in \text{Modif}(\delta)$, then the $A_i$-value of $t \in I$ is replaced by a new distinct variable not occurring elsewhere. Note that $\delta_{\alpha}(I)$ is determined (modulo variable renaming) by $I$ and $\text{Modif}(\delta)$. For $\delta_1$ and $\delta_2$ as above, we obtain $\delta_1_{\alpha}(\text{EMP}) = F_1$ and $\delta_2_{\alpha}(\text{EMP}) = F_2$. Then, it is tempting to think that for each set-minimal repair $\delta$ of $I$ under $\Sigma$, $\delta_{\alpha}(I)$ is a fix and $\delta(I)$ an uprepair.

Nevertheless, [7] contains an example where the apparent relationship between set-minimal repairs and uprepairs breaks down. Let $K$ the following relation subject to the constraints at the right.

$$K | A \ B \ C \quad \forall X (\neg R(X, b, c))$$
$$\quad a \ b \ c \quad (r) \quad \forall Y (\neg R(a, Y, d))$$

Consider the following updates, each of which renders $K$ consistent:

$$\delta_3 = \{((r, A), a'), ((r, C), c'), ((s, B), b'), ((s, C), d')\}$$
$$\delta_4 = \{((r, B), b'), ((s, A), a')\}$$
$$\delta_5 = \{((r, B), b'), ((r, C), d'), ((s, A), a'), ((s, C), c')\}$$
Hence,

\[
\text{Modif}(\delta_3) = \{(r; A), (r; C), (s; B), (s; C)\}
\]
\[
\text{Modif}(\delta_4) = \{(r; B), (s; A)\}
\]
\[
\text{Modif}(\delta_5) = \{(r; B), (r; C), (s; A), (s; C)\}
\]

Both \(\delta_3\) and \(\delta_4\) are set-minimal repairs. Since \(\text{Modif}(\delta_4) \subseteq \text{Modif}(\delta_5)\), the update \(\delta_5\) is not a set-minimal repair, even though \(\delta_5(K) = \delta_3(K)\). That is, an update that is not a set-minimal repair can yield the same relation as some set-minimal repair.

\[
\delta_3(K) = \delta_5(K) = \delta_3^\text{var}(K) = \delta_4^\text{var}(K)
\]

The tableau \(\delta_4^\text{var}(K)\) is a fix. The substitution \(\theta = \{X/X, Y/Y, Z/d, W/c\}\) is a one-one homomorphism from \(\delta_3^\text{var}(K)\) to \(\delta_4^\text{var}(K)\), showing that \(\delta_3^\text{var}(K)\) is not a fix. Consequently, \(\delta_3\) is a set-minimal repair, but \(\delta_3^\text{var}(K)\) is not a fix and \(\delta_3(K)\) is not an uprepair. We believe that in this example, uprepairs behave more naturally than set-minimal repairs because \(\delta_4(K)\) looks closer to \(K\) than \(\delta_3(K)\).

References

Some Research Directions in Consistent Query Answering: A Vision

Leopoldo Bertossi*  
Carleton University, School of Computer Science  
Ottawa, Canada  
bertossi@scs.carleton.ca

Some Past Research

Research in consistent query answering (CQA) in databases was initiated in the database community with the publication of [1], where the main goal was to formalize the notion of consistent answer to a query posed to a possibly inconsistent database, i.e. that fails to satisfy a given set of integrity constraints (ICs) that are not enforced by the system.

For many reasons [9], such inconsistencies may naturally arise. Consistent answers were semantically characterized as those answers that can be obtained, as normal answers, from all the possible minimally repaired versions of the inconsistent database at hand. According to [1], a repair of a relational database instance $D$ is an instance that satisfies the ICs, with the same schema as $D$, that in set theoretic terms, minimally differ from $D$ wrt whole tuples that are either deleted or inserted in order to restore consistency. Then, the consistent answers to a query are those that are invariant under repairs.

Since computing consistent answers by appealing directly to the definition, i.e. via explicit computation and querying of all possible repairs, is [9] practically unfeasible, [1] introduced the first mechanism for computing consistent answers to first-order queries that did not appeal to explicit computation of repairs. Instead, the idea was to modify the original query without changing the inconsistent database, then pose the rewritten query, and collect the normal answers to it. This mechanism turned out to be sound and/or complete for several classes of queries and ICs that were identified in [1]. This form of query rewriting works for some useful classes of queries and ICs, but its applicability is still limited.

With the purpose of computing consistent answers to full first-order queries, in [5, 19] the repairs of a database were characterized as the stable models of disjunctive logic programs. In consequence, obtaining consistent answers became the problem of computing answers from the repair program combined with the query program under the skeptical stable model semantics. Since the query program is quite general, the queries supported could be found much beyond first-order, actually in rich extensions of Datalog. However, the number of atoms in the ICs may produce a blow-up in the number of program rules.

The semantics of consistent query answers to scalar aggregate queries wrt functional dependencies was introduced and studied in [3]. Since those queries

* Also Faculty Fellow of the IBM Center for Advanced Studies (Toronto Lab.)
return numerical values, the natural semantics is the *range semantics*, i.e. based on the shortest numerical interval that contains the answers to the query from all possible repairs. Algorithms were given to compute the range semantics for the (provably) tractable cases, and the untractable cases were fully characterized [4].

The semantics of CQA was further studied in a non-classical logical system, the *annotated predicate logic* (APC). As a result, database repairs were characterized as a special class of minimal models of a theory written in a particular version of APC (the truth-lattice is a parameter in APC, so the right lattice for this task was identified). This turned the problem of obtaining consistent answers to an arbitrary first-order query into a problem of non-monotonic reasoning from an APC theory [apc,nmr], for which there are no implementations available.

The results obtained using answer set programming and the theoretical framework obtained via APC motivated trying to mimic APC reasoning using answer set programming. So, the truth annotations introduced in [2] were used as distinguished domain constants to be used in database atoms extended with an extra argument. Reasoning with annotations and implicit interaction between them in the APC version was made explicit in answer set programming [7, 8, 12].

In this way, it is possible to capture every IC, no matter how many atoms in it, as a single rule in the program, avoiding the exponential blow-up [5]. The stable models of the generated disjunctive logic program (with program denial constraints) were established to be in one-to-one correspondence with the repairs [8]. So, CQA could be done for any kind of universal ICs (i.e. no existential quantifiers in it) and any query expressed in extensions of Datalog.

In [12] referential ICs (they contain existential quantifiers) are considered, and the programs introduced in [7] were extended accordingly, assuming that referential ICs could be repaired through tuple deletions or insertions of null values that are not propagated through other ICs. The other edge of the problem, also addressed in [12], but never considered before, is that the original database could already be *incomplete*, i.e. containing null values. In consequence, their presence has to be considered when defining repairs.

In [8, 14] it is shown how the logic programming approach to CQA can be made more efficient by applying several optimizations, like pruning unnecessary program rules, rule transformation to capture cases of lower complexity (e.g. head-cycle-free), optimizing query evaluation using *magic sets* techniques, optimizing the access to the underlying database, etc.

In between, a much more clear picture of the complexity of CQA has emerged. Also tractable classes have been identified and implementations developed [15, 13, 18]. To the best of our knowledge, all the algorithms and implementations available compute consistent answers to queries from scratch, except for the possible precomputation of all the repairs (or stable models in the case of logic programs).
Looking Forward

Many problems are still open in the area of consistent query answering. Many of them, interesting, challenging, but also specific to particular techniques, applications, variations of the basic notions, and implementation. However, a general and still open problem is to achieve a global understanding of the “logic of consistent query answering”, i.e. the logic that governs the definition and computation of consistent query answers, and reasoning with and about them.

We know, e.g. that CQA follows a non-monotonic logic [9]. We also know that it is, in some sense, a form of modal logic (being the repairs the possible worlds). We can see that it is not compositional as classical query answering in databases, e.g. the answer set to a conjunctive query may not be the intersection of the answer sets, etc. We do not have a complete knowledge of its logic or their properties. In particular, compositionality of CQA has not been investigated. We do not know what is correct or what can be used for query answering in that direction yet. This is an objective that deserves much research.

Let us recall that classical query answering in relational databases follows, essentially, a first-order logic, whose most notorious expression is the relational calculus. From this point of view, it is semantically clear what is an answer to a query and how answers to queries can be combined in order to give answers to more complex queries. This is because, the notion of truth in first-order logic has nice compositional properties, as established by its Tarskian semantics. Already the above mentioned non-monotonicity makes CQA depart from first-order logic, that is monotonic.

Non-monotonic formalisms have been used to characterize and compute CQA, e.g. annotated predicate logic [2], logic programs with stable model semantics [5, 7], non-monotonic analytic tableaux [10], circumscription [10]. However not much emphasis has been placed on the study of the intrinsic logic of consistent query answering. There are natural open questions in this direction: (a) How can we classify the underlying non-monotonic logic? (b) What kind of modal logic we have? With what kind of accessibility relation? Can it be axiomatized? (c) What compositionality properties it has? (d) Is there a set-theoretic, algebraic counterpart (like relational algebra to relational calculus)? (d) What are the connections to other logics that have been used to capture and formalize ICs in databases. Addressing all these issues becomes more interesting and difficult if one considers that the database may be incomplete, and the database community, including database practice, is far from having an agreement on the semantics of incomplete databases.

The repair semantics that has been intensively studied is the one introduced in [1]. However, other repair semantics have been considered: Repairs that minimally differ in cardinality from the original database [9, 5], and repairs that minimize some aggregation function over the differences of attribute values between the repair and the original instance [17, 20, 11, 16]. It should be clear that any choice of a repair semantics will have an effect on the underlying logic of CQA. We think that identifying general properties of the reasonable repair semantics and studying their impact on the logic of CQA is a very important re-
search direction. Unifying principles seem to be necessary at this stage, in order to have a better understanding of CQA, both in theoretical and practical terms.

References


About queries addressed to possibilistic databases

Patrick Bosc and Olivier Pivert

IRISA-ENSSAT/Université de Rennes 1
{pivert,bosc}@enssat.fr

Abstract. In this position paper, relational databases where some attributes may take imprecise values represented by possibility distributions, i.e., weighted sets of candidate values, are considered. The main objective is to point out classes of queries that can be used to retrieve information from such databases, with the two following constraints: the queries must have a sound semantics and the complexity of their evaluation must be kept reasonable.

1 Introduction

Some well-known works conducted in the 80s have revealed that the presence of null values (in the sense of existing unknown values) in databases raises several serious problems. In particular, it is not feasible to extend the relational operators, defined for precise and complete relations, so as to make them work on relations containing imprecise information. Indeed, when null values are involved, the result of some operations (such as the join) cannot be represented by means of a “basic” relational table (a more sophisticated model is needed). Now, a null value only constitutes a special case of disjunction where no preferences are expressed over the candidates (which are exclusive). Therefore, the same problem obviously arises in the case of more elaborate imprecise values such as possibility distributions. It is thus important to identify classes of queries that can be used in a such database context. After a short presentation of the main characteristics of a possibilistic database, two main lines of research are successively reported: a restricted algebra for possibilistic databases which serves as a basis to diverse forms of queries and possibilistic queries of the type “to which extent is it possible that tuple t belongs to the answer Q?” (where t is explicitly specified and Q is a query calling on some algebraic operators).

2 Possibilistic relational databases

Possibility theory [8] provides an ordinal model for uncertainty where imprecision is represented by means of a preference relation coded by a total order over the possible situations. This framework is closely related to fuzzy set theory since the idea is to constrain the values taken by a variable thanks to a (normalized) fuzzy set called a possibility distribution. A possibility distribution is an application \( \pi \) from a domain \( X \) to the unit interval \([0, 1]\) and \( \pi(a) \) expresses the degree to which \( a \) is a possible value for the considered variable. The normalization condition imposes that at least one of the values of the domain \( (a_0) \) is completely possible, i.e., \( \pi(a_0) = 1 \). This
setting is particularly suited to take into account uncertainty represented by
linguistic terms such as "high", "large", "expensive" and so on. Only finite
possibility distributions are considered here and they are written \( \{\pi_1/a_1, \ldots, \pi_n/a_n\} \)
where \( a_i \) is a candidate value and \( \pi_i \) its possibility degree. Any event \( E \) defined on
the powerset of \( X \) is characterized by two measures \( \Pi \) and \( N \). The axioms related to the
measure of possibility \( \Pi \) are the following: i) \( \Pi(X) = 1 \) (which requires the
normalization condition), ii) \( \Pi(\emptyset) = 0 \) and iii) \( \Pi(E_1 \cup E_2) = \max(\Pi(E_1), \Pi(E_2)) \). The
measure of possibility of the event \( E \) is derived from the possibility distribution
associated with the concerned variable in the following way: \( \Pi(E) = \max_{x \in E} \pi(x) \). The
possibility of the conjunction of two non interactive events is given by: \( \Pi(E_1 \land E_2) = \min(\Pi(E_1), \Pi(E_2)) \).

The only relationship between the possibility of \( \bar{E} \) (the opposite event of \( E \)) and
that of \( E \) is: \( \max(\Pi(E), \Pi(\bar{E})) = 1 \), which entails that if \( \Pi(E) = 1 \), \( \Pi(\bar{E}) \) can range from 0
to 1. To better characterize \( E \), the measure of certainty (or necessity) \( N \) has been also
introduced: \( N(E) = 1 - \Pi(\bar{E}) \). In other words, the less possible \( \bar{E} \), the more certain \( E \).
As far as regular (i.e., non-fuzzy) events are concerned, it can be proven that: \( \Pi(E) < 1 \Rightarrow N(E) = 0 \).
Then, these two measures provide a total order over the set of events
which can be ordered according to \( \Pi \) for those who are not at all certain and
according to \( N \) for those which are completely possible.

In a possibilistic relational database \( D \), some attribute values are imprecisely
known and then represented as possibility distributions. It is worth mentioning that
the possibilistic framework proposes a unified framework for representing precise
values, as well as imprecise ones (or-sets) or fuzzy ones (the most general case),
including the "null values" (see e.g. [1]). The first version of a possibilistic database
model was introduced by Prade in the mid 80s [7].

From a semantic point of view, a possibilistic database \( D \) can be interpreted as a
set of usual databases (also called worlds) \( W_1, \ldots, W_p \), denoted as \( \text{rep}(D) \), each of
which being more or less possible. This view establishes a strong connection
between possibilistic and regular databases. It is particularly interesting since it
offers a canonical approach to the interpretation of queries addressed to possibilistic
databases. Any world \( W_i \) is obtained by choosing a candidate value in each
possibility distribution (i.e., by choosing an element in the weighted set
corresponding to the possibility distribution) appearing in \( D \). It corresponds to a
conjunction of independent choices and its degree of possibility is the minimum of
the degrees tied to each of the chosen candidate values in \( D \).

**Example.** Let us consider the possibilistic database \( D \) involving two relations: \( \text{im} \)
and \( \text{pl} \) whose respective schemas are \( \text{IM}(\#i, \text{ac}, \text{date}, \text{pl}) \) and \( \text{PL}(\text{ac}, \text{lg}, \text{msp}) \).

\[
\begin{array}{|c|c|c|c|}
\hline
\text{im} & \#i & \text{ac} & \text{date} & \text{pl} \\
\hline
i1 & a1 & \{1/d1, 0.7/d3\} & c1 & \\
\hline
i3 & \{1/a3, 0.4/a4\} & d1 & c2 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{pl} & \text{ac} & \text{lg} & \text{msp} \\
\hline
a1 & 20 & 1000 & \\
a2 & 25 & 1200 & \\
a3 & 18 & 800 & \\
a4 & 20 & 1200 & \\
\hline
\end{array}
\]
Relation im describes satellite images of aircrafts identified by a number (#i), a location (pl) and the date of the shot (date) which are supposed to include a single aircraft (ac). Relation pl gives the length (lg) and maximal speed (msp) of each aircraft. With the extensions of im and pl given before, four worlds can be drawn, $W_1$, $W_2$, $W_3$ and $W_4$, since there are two candidates for date in the first tuple of im and two candidates for ac in the second one. Each world involves the relation pl which has only precise values (and then has a single interpretation) and one among four regular relations $im_1$ to $im_4$, issued from the possibilistic relation im. Two such relations are:

<table>
<thead>
<tr>
<th></th>
<th>#i</th>
<th>ac</th>
<th>date</th>
<th>place</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>a1</td>
<td>d3</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>i3</td>
<td>a3</td>
<td>d1</td>
<td>c2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>#i</th>
<th>ac</th>
<th>date</th>
<th>place</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>a1</td>
<td>d3</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>i3</td>
<td>a4</td>
<td>d1</td>
<td>c2</td>
<td></td>
</tr>
</tbody>
</table>

which are respectively possible at the degrees 0.7 and 0.4.

3 A restricted relational algebra

A first line of research deals with extending algebraic queries in the presence of imprecise data, i.e., with defining the algebraic operators that may be part of a querying language in a possibilistic database context. Let us first remark that the number of worlds associated with a possibilistic database may be huge even when the number of ill-known values in the possibilistic database is relatively small. A crucial objective is thus to define a “compact” processing method, i.e., a method that does not require to make the interpretations of the database explicit. Consequently, it is mandatory to design a data model that constitutes a strong representation system for the query language considered, i.e., a model that supports a closed set of operations whose results are consistent with a world-based interpretation. In other terms, if $\text{rep}(D)$ denotes the set of worlds associated with the possibilistic database $D$, the following property must hold for any operation $o$ of the language: $o(\text{rep}(D)) = \text{rep}(o(D))$. Such a framework guarantees a sound semantics for the operators of the language, and it makes it possible to envisage a tractable query evaluation process.

It turns out that the “basic” possibilistic relational model (i.e., the regular relational model extended to the case where an attribute value may be a possibility distribution) is not even a strong representation system for the selection. Therefore, an extended possibilistic database model has been devised, which involves usual possibilistic relations enriched with: i) a degree $N$ associated with every tuple, which expresses the certainty that the tuple has a representative in any interpretation of the relation (or, reciprocally, the possibility for the tuple to have no representative in a world derived from the considered possibilistic relation), ii) the introduction of nested relations used to represent the dependencies between candidate values for different attributes and thus to model possibility distributions over several attributes.

In [2], it is shown that this extended model constitutes a strong representation system for the operations of selection, projection, foreign key join and union. The last two operators are interesting inasmuch as they allow for combining relations. In particular, the specific join operator makes it possible to compose a possibilistic relation where the join attribute(s) may take ill-known values with a usual relation.
modeling a functional dependency. For example, let us consider a database including
a regular relation representing the functional associations between car names and car
brands and a possibilistic one describing car purchases for which the name of the car
that has been purchased may be imprecise. The query that aims at retrieving the
persons who have purchased a car of a given brand entails joining the two tables.
Processing such a join involves determining the image of an ill known value by
means of the table associated with the function. The resulting table includes a nested
relation whose purpose is to store the pairs of values linked by the function, as well
as the corresponding possibility degrees.

Let us mention that a key point lies in the fact that the extended operators are
fairly close to those applying to regular databases, which lets expect reasonable
performances as long as convenient ways for implementing imprecise values can be
found out when a usual DBMS is used. Moreover, the approach proposed can be
straightforwardly adapted to probabilistic databases, i.e., where imprecise attribute
values are represented as probability distributions.

One may wonder about the practical interest of a result made of an extended
possibilistic relation. So, more user-oriented queries, called extended yes/no queries,
have been studied whose general form is: “to what extent is it possible and certain
that the answer to Q fulfills property P?”. Of course, Q is restricted to the operators
pointed out before to make sure that it can be processed in a compact fashion, i.e.,
directly on (extended) possibilistic relations. Diverse types of property P are
investigated, such as “contains tuple t”, or “is [not] empty”, or “contains at most k
tuples”. Processing such queries calls on a two-step procedure: i) the algebraic query
Q is evaluated in a compact way and ii) a post processing (in general based on a “trial
and error” technique along with pruning conditions) takes place in order to assess
the satisfaction of property P over the possibilistic relation obtained.

4 Possibilistic queries

Possibilistic queries are of the form: “to which extent is it possible that tuple t
belongs to the answer to query Q?” where Q is a relational query. They can be seen
as a weakened form of a family of queries considered previously where the user is
only interested in the possibility of events, not their necessity. One way for process-
ing such queries is to rely on the approach outlined before, whose main interest is
to guarantee the tractability of the process. But, there is a counterpart in the sense
that the query Q is strongly constrained. From that, one may wonder about the feasi-
ibility of allowing a wider range of operations in Q, while keeping a tractable process.

Here, since we are only interested in the presence of a given tuple in the result, it
is not necessary to have available a strong representation system. It is indeed
sufficient to consider the tuples that may appear in the union of all the worlds
associated with the considered database. The principles of the evaluation procedure
and the definition of some of the extended operators are reported in [2]. Beyond this,
we have: i) proven the correctness of the approach, ii) identified the family of queries
that can be evaluated using this technique in a compact way, i.e., in the context of the
“basic” possibilistic database model and iii) defined the extended versions of all the
operators authorized in such queries. It appears that the solution proposed is suited
to any algebraic query Q that does not involve any difference operation and that
does not include several references to a same relation. The case of possibilistic
queries involving difference operations has been recently studied and the interested reader will find details in [5].

5 Conclusion and perspectives

Databases containing imprecise information have been considered. Some attributes may take values which are described as weighted sets of candidates in the context of possibility theory which is an alternative framework with respect to the more common probabilistic setting. Diverse types of queries of interest in the context of possibilistic databases have been outlined. As to algebraic queries, it turns out that the only tractable queries are those where a limited set of operations is allowed. However, as mentioned before, it is worth noticing that the approach taken for possibilistic databases can be adapted to databases where imprecise attribute values are described using probability distributions (the basic probabilistic model must be extended to multiple-attribute probability distributions and the operators must take into account the additive nature of probability theory). On the contrary, it has been shown that the processing strategy devised for the family of queries known as possibilistic queries (“to what extent is it possible that tuple t belongs to the result of Q” where Q may involve any operator except the difference) cannot be adapted to queries of the form “to what extent is it probable that tuple t belongs to the result of Q”.

One important line for future works concerns implementation and experiments in order to measures the actual performances reached by prototypes of database management systems able to support databases with possibilistic information and non-trivial queries against them.

References