International Journal of Approximate Reasoning 55 (2014) 501–528

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar

Policy-based inconsistency management in relational databases

Maria Vanina Martinez[a], Francesco Parisi[b], Andrea Pugliese[b,*], Gerardo I. Simari[a], V.S. Subrahmanian[c]

[a] Department of Computer Science, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
[b] Università della Calabria, Via Bucci, 87036 Rende CS, Italy
[c] University of Maryland College Park, College Park, MD 20742, USA

ARTICLE INFO

Article history:
Received 24 June 2013
Received in revised form 26 November 2013
Accepted 9 December 2013
Available online 19 December 2013

Keywords:
Inconsistency management
Relational databases

ABSTRACT

Though inconsistency management in databases and AI has been studied extensively for years, it does not allow the user to specify how he wants to resolve inconsistencies. In real-world applications, users may want to manage or resolve inconsistencies based not only on the data, but their own knowledge of the risks involved in decision making based on faulty data. Each user should be empowered to use reasonable policies to deal with his data and his mission needs. In this paper, we start by providing an axiomatic definition of *inconsistency management policies* (IMPs) that puts this power in the hands of users. Any function satisfying these axioms is an IMP. We then define three broad families of IMPs, and derive several results that show (*i*) how these policies relate to postulates for the revision of *belief bases* and to recent research in the area of *consistent query answering*, and (*ii*) how they interact with standard relational algebra operators. Finally, we present several approaches to efficiently implement an IMP-based framework.

1. Introduction

Reasoning with or in the presence of inconsistent knowledge bases has been the focus of a vast amount of research in AI for over 30 years. Logics of inconsistency were introduced in the 1960s, and were later developed [1–8]. The work of [2] introduced a four valued logic that was used for handling inconsistency in logic programming [4] and extended to the case of bilattices [9]. Frameworks such as default logic [10], maximal consistent subsets [11], inheritance networks [12], and others were used to generate multiple plausible consistent scenarios (or “extensions”), and methods to draw inferences were developed that examined truth in all (or some) extensions. [13] extended annotated logics of inconsistency developed in [4] to handle a full first order case, while [14] developed similar extensions to handle inheritance networks.

Syntactic approaches have also been adopted such as the one discussed in [15]; finally, argumentation methods [16–20] have been used for handling uncertainty and inconsistency by means of reasoning about how certain contradictory arguments defeat each other. Other important contributions include [21], and studies in the database community include [11, 22–27]. These, and other work on consistent query answering, focused on the problem of extracting “reliable” information from inconsistent data, and described possible ways of repairing a database so that consistency is preserved (a comprehensive survey of the work done on consistent query answering can be found in [28]). Important work has also been done

* Corresponding author.
E-mail addresses: vanina.martinez@cs.ox.ac.uk (M.V. Martinez), fparisi@dimes.unical.it (F. Parisi), andrea.pugliese@unical.it (A. Pugliese), gerardo.simari@cs.ox.ac.uk (G.I. Simari), vs@umiacs.umd.edu (V.S. Subrahmanian).

0888-613X/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ijar.2013.12.004

on the use of logic programs to specify repairs and adopt them for logic-programming based query answering [29–37]. [38] argued that inconsistency can often be resolved in different ways based on what the user wants, and they provided a mechanism to reason about maximal consistent subsets (also called "repairs" by others such as [23]) using objective functions that the user gets to choose.

The work referred to above does not take into account the fact that there are many settings where the end-user of a database application needs to bring his knowledge of the application to bear when handling inconsistency. A database administrator for the astronomy department may know (for example) the general guidelines for how to design an astronomy database (schemas, keys, etc.); however, he is not likely to know how such data was collected, what assumptions were made when collecting it, and how inconsistencies should be handled[1] so as to meet individual astronomers' needs. In the same vein, a database developer/administrator for major DB service providers may not understand how some epidemiological data in a government's Health Ministry was collected, and what the ramifications and risks are to a policy maker using that data to make decisions. The policy-maker may wish to make decisions on the basis of some inconsistent data taking into account not only information about how the data was collected, which sources were reliable, and so forth, but also about his own risk if he makes the wrong decision. A database administrator who has embedded some default form of inconsistency management within the DB infrastructure will not know this a priori.

The goal of this paper is to develop the theory and implementation support required in databases so that *end users* (*such as the two mentioned above*) *can bring both their application specific knowledge as well as their own personal risk to bear when resolving inconsistencies*. To reiterate why this is important, let us consider the following example.

Example 1. The *Emp* relation below represents data about employees, their salaries and tax brackets; each tuple t_i in the relation is provided by a source s_j (which is annotated on the same row).

	Name	Salary	Tax_bracket	*Source*
t_1	John	70K	15	s_1
t_2	John	80K	20	s_2
t_3	John	70K	25	s_3
t_4	Mary	90K	30	s_1

Let us assume that salaries are uniquely determined by names. In this case, a user may want to resolve the inconsistency about John's salary in many different ways.

C1 If he were considering John for a loan, he might want to choose the lowest possible salary of John to base his loan on. Here, the user's decision is based on the *risk* he would take if he gave John a loan amount based on a higher income.
C2 If he were assessing the amount of taxes John has to pay, he may choose the highest possible salary John may have—in this case, the end user is basing his decision on his own *reward* (assuming his salary is somehow based on the amount of additional taxes he can collect).
C3 If he were just trying to estimate John's salary, he may choose some number between 70K and 80K (*e.g.*, the average of the three reports of John's salary) as the number.
C4 If he has different degrees of confidence in the sources that provided these salaries, he might choose a weighted mean of these salaries.
C5 He might choose not to resolve the inconsistency at all, but to just let it persist until he can clear it up—there may be no need to deal with John's salary with respect to his current task.
C6 He might simply consider *all* the data about John unreliable and might want to ignore it until it can be cleared up—this is the philosophy of throwing away all contaminated data. This is more likely to happen, for example, when there is a scientific experiment with inconsistent data or when there is a critical action that must be taken, but cannot be taken on the basis of inconsistent data.

Each of cases C1 through C6 reflects a *policy* that a user (or different users of the same database) is (are) using to resolve inconsistencies within the same database. In this paper, we extend the classical relational algebra with the notion of *inconsistency management policies* (IMPs for short). IMPs are functions that specify how inconsistency must be handled—and these IMPs can be specified directly by a user (or chosen/customized by the user from a suite of general policy "templates" provided in a user menu) when he is asking a query. Thus, *how* to resolve an inconsistency is specified by the user. In contrast, important work on data cleaning and consistent query answering has been developed for inconsistency management in relational databases; the first approach describes possible ways of repairing a database so that consistency is preserved or restored; the latter aims to characterize meaningful answers from possibly inconsistent databases, *i.e.*, those answers to a

[1] We use the word "handled" informally in the Introduction to explain what needs to be done with inconsistent data. Many approaches have been proposed in the literature (including that in this paper). The phrase "handle inconsistency" refers to any method that claims to repair, revise, or reduce inconsistent information in a database.