UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Community-oriented information integration

Permalink
https://escholarship.org/uc/item/83k4z81v

Author
Katsis, Ioannis

Publication Date
2009

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Community-oriented Information Integration

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Ioannis Katsis

Committee in charge:

Professor Alin Deutsch, Co-Chair
Professor Yannis Papakonstantinou, Co-Chair
Professor Richard Belew
Professor Bertram Ludaescher
Professor Victor Vianu

2009
The dissertation of Ioannis Katsis is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2009
DEDICATION

To my family
EPIGRAPH

*We cannot learn without pain*

− Aristotle
TABLE OF CONTENTS

Signature Page ................................................................. iii
Dedication ................................................................. iv
Epigraph ................................................................. v
Table of Contents ................................................................. vi
List of Figures ................................................................. ix
Acknowledgements ................................................................. xi
Vita and Publications ................................................................. xiv
Abstract of the Dissertation ................................................................. xv

Chapter 1  Introduction ................................................................. 1
   1.1 Current State of the Art: Enterprise Information Integration ................................................................. 4
   1.2 Our Proposal: Community-oriented Information Integration ................................................................. 5
      1.2.1 Source Registration ................................................................. 9
      1.2.2 Inconsistency Resolution ................................................................. 11
   1.3 Outline ................................................................. 13

Chapter 2  Source Registration ................................................................. 15
   2.1 Background ................................................................. 15
   2.2 Overview of RIDE ................................................................. 17
      2.2.1 Contributions ................................................................. 19
   2.3 Running Example ................................................................. 21
   2.4 Mapping Interfaces ................................................................. 22
   2.5 RIDE Interaction ................................................................. 24
      2.5.1 Suggestion Component ................................................................. 24
      2.5.2 Inconsistency Component ................................................................. 30
      2.5.3 Dealing with Parameterized Queries ................................................................. 32
      2.5.4 RIDE’s Properties ................................................................. 32
   2.6 Formal Specifications ................................................................. 33
      2.6.1 Community-oriented Integration ................................................................. 33
      2.6.2 Inconsistency ................................................................. 36
      2.6.3 Levels of Self-Reliance ................................................................. 37
   2.7 Algorithms ................................................................. 39
      2.7.1 Deciding Inconsistency ................................................................. 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.2 Deciding Self-Reliance Levels</td>
<td>44</td>
</tr>
<tr>
<td>2.7.3 Computing Suggestions</td>
<td>45</td>
</tr>
<tr>
<td>2.7.4 Complexity</td>
<td>47</td>
</tr>
<tr>
<td>2.7.5 Experimental Evaluation</td>
<td>49</td>
</tr>
<tr>
<td>2.8 Related Work</td>
<td>51</td>
</tr>
<tr>
<td>2.9 Extensions and Future Work</td>
<td>52</td>
</tr>
<tr>
<td>2.9.1 Supporting Evolution of Communities</td>
<td>52</td>
</tr>
<tr>
<td>2.9.2 Contributing to a List of Queries</td>
<td>53</td>
</tr>
<tr>
<td>2.10 Conclusions</td>
<td>54</td>
</tr>
<tr>
<td>Chapter 3 Self-Reliance Levels Revisited</td>
<td>55</td>
</tr>
<tr>
<td>3.1 Background</td>
<td>55</td>
</tr>
<tr>
<td>3.2 Preliminaries</td>
<td>58</td>
</tr>
<tr>
<td>3.3 Self-Reliance Levels</td>
<td>61</td>
</tr>
<tr>
<td>3.4 Examples</td>
<td>64</td>
</tr>
<tr>
<td>3.5 Main Results</td>
<td>69</td>
</tr>
<tr>
<td>3.5.1 Self-Sufficient Registrations</td>
<td>70</td>
</tr>
<tr>
<td>3.5.2 Now-Complementary Registrations</td>
<td>71</td>
</tr>
<tr>
<td>3.5.3 Later-Complementary Registrations</td>
<td>72</td>
</tr>
<tr>
<td>3.5.4 Non-recursive Certain Rewritings</td>
<td>75</td>
</tr>
<tr>
<td>3.6 Related Work</td>
<td>82</td>
</tr>
<tr>
<td>3.7 Conclusions</td>
<td>83</td>
</tr>
<tr>
<td>Chapter 4 Inconsistency Resolution</td>
<td>84</td>
</tr>
<tr>
<td>4.1 Background</td>
<td>84</td>
</tr>
<tr>
<td>4.2 System’s Overview</td>
<td>86</td>
</tr>
<tr>
<td>4.3 System’s Architecture</td>
<td>89</td>
</tr>
<tr>
<td>4.4 Data Model</td>
<td>91</td>
</tr>
<tr>
<td>4.4.1 Definition</td>
<td>92</td>
</tr>
<tr>
<td>4.4.2 Semantics</td>
<td>96</td>
</tr>
<tr>
<td>4.4.3 Ac-Data Manipulation Language</td>
<td>97</td>
</tr>
<tr>
<td>4.4.4 Comparison to other data models</td>
<td>99</td>
</tr>
<tr>
<td>4.5 Resolution Actions &amp; Policies</td>
<td>100</td>
</tr>
<tr>
<td>4.5.1 Resolution Actions</td>
<td>101</td>
</tr>
<tr>
<td>4.5.2 Resolution Policies</td>
<td>103</td>
</tr>
<tr>
<td>4.6 Query Answering</td>
<td>106</td>
</tr>
<tr>
<td>4.7 Resolution Actions on Query Results</td>
<td>109</td>
</tr>
<tr>
<td>4.8 Implementation</td>
<td>110</td>
</tr>
<tr>
<td>4.8.1 Flattening Ac-Relations</td>
<td>111</td>
</tr>
<tr>
<td>4.8.2 Nesting Flat Relations</td>
<td>114</td>
</tr>
<tr>
<td>4.8.3 Answering Queries</td>
<td>115</td>
</tr>
<tr>
<td>4.8.4 Executing Resolution Policies</td>
<td>118</td>
</tr>
<tr>
<td>4.9 Related Work</td>
<td>119</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Enterprise Information Integration Architecture</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Community-oriented Information Integration Architecture</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Screenshot from an interaction with RIDE</td>
<td>10</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Screenshot from an interaction with Ricolla</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Community-oriented Integration Architecture</td>
<td>18</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Source schemas and Target schema</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Application query</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Barnes &amp; Noble Registration</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Sample interaction for the B &amp; N registration</td>
<td>26</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Sample interaction for the Prentice Hall registration</td>
<td>31</td>
</tr>
<tr>
<td>Figure 2.7</td>
<td>Canonical Source and Target Instance</td>
<td>41</td>
</tr>
<tr>
<td>Figure 2.8</td>
<td>Experimental Setting</td>
<td>50</td>
</tr>
<tr>
<td>Figure 2.9</td>
<td>RIDE’s response times w.r.t. # of registered sources</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Decidability of categorizing registrations (see Section 3.5 for complexity results)</td>
<td>58</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Source Mappings</td>
<td>63</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Registrations for Example 10</td>
<td>73</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Schema of Movie Ac-Database</td>
<td>86</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Ac-relation Actor containing a single ac-tuple for Clint Eastwood</td>
<td>87</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Ricolla Architecture</td>
<td>90</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Ac-relation Movie</td>
<td>93</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Ac-relation MovieActor</td>
<td>93</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Ac-tuples corresponding to movies of actors with a fan mailing address in Burbank</td>
<td>94</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Interpretations of the ac-tuple in Figure 4.2</td>
<td>97</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>‘User Actions’ table for ‘1.85m’ ac-alternative in Clint Eastwood’s ac-tuple</td>
<td>98</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>General form of a Resolution Policy</td>
<td>103</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Default Resolution Policy for Lara</td>
<td>105</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Flat representation of the ac-tuple in Figure 4.2 through the ac-tuple’s interpretations</td>
<td>112</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Flat representation of the ac-tuple in Figure 4.2 using the algorithm Flatten</td>
<td>113</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>Query Answering Semantics and Implementation</td>
<td>116</td>
</tr>
<tr>
<td>Figure A.1</td>
<td>Proof of Theorem 2: Auxiliary Integration System $IS_{aux}$: Schemas, Assertions, Source Constraints and Target Constraints</td>
<td>129</td>
</tr>
<tr>
<td>Figure A.2</td>
<td>Proof of Theorem 2: Auxiliary Integration System $IS_{aux}$: Mapping Constraints</td>
<td>130</td>
</tr>
</tbody>
</table>
Figure A.3: Proof of Theorem 2: Integration System IS used in the Reduction

Figure A.4: Proof of Theorem 6: Self Sufficiency Problem Instance for First
  Cut: Schemas and Target Constraints ........................................ 132

Figure A.5: Proof of Theorem 6: Self Sufficiency Problem Instance for First
  Cut: Mappings and Application Query ........................................ 133

Figure A.6: Proof of Theorem 6: Self Sufficiency Problem Instance for Sec-
  ond Cut: Schemas and Target Constraints ................................. 135

Figure A.7: Proof of Theorem 6: Self Sufficiency Problem Instance for Sec-
  ond Cut: Mappings and Application Query ................................. 136
ACKNOWLEDGEMENTS

Coming to the end of the amazing journey that led to this dissertation, I would like to thank everybody that supported, encouraged and guided me along the way.

First of all, I am grateful to both my advisors, Alin Deutsch and Yannis Papakonstantinou, for teaching me how research is conducted and for creating a unique combination that exposed me to database research from both the theoretical and the systems’ perspective. Alin taught me patiently the chase, spent countless hours discussing the technical details of our work and encouraged me when needed. Yannis inspired me with his passion for systems, showed me over and over how to design a useful system and introduced me to commercialization and large project management. They both taught me how to write and talk (in research terms at least) and they provided advice and help at countless occasions that would require more pages than the reader of this text would be willing to read to list comprehensively. Choosing a research topic, getting an internship and deciding what to do after graduation are just a few moments where their help proved invaluable. However, most importantly I would like to thank them for being not just advisors but also very good friends, who went beyond the strict advisor-advisee relationship and helped me feel right at home from the first time I met them.

Moreover, I would like to thank the remaining three members of my thesis committee for their help and support during my PhD: Victor Vianu, Richard Belew and Bertram Ludaescher. Victor, not only for being the first to introduce me to database theory through his graduate course but also for his helpful comments and suggestions during my presentations at the database seminar and for his immense help during my job search. Richard, who, by having a broader background than just databases, offered a very much needed higher-level perspective of my research and who reminded me of the reality check that we all need to do when we work on a problem for a long amount of time. Finally, Bertram for his constructive comments during my thesis proposal, for his enthusiasm on integration-related research and for his encouragement to continue working on it.
Thanks are also due to all my collaborators, who went out of their way to help me and guide me. Phil Bernstein and Sergey Melnik for giving me the opportunity to spend two great summers at Microsoft Research, for devoting a large amount of time to assist me and for their excellent feedback on my write-ups. Mauricio Hernandez and the Clio Group led by Howard Ho for an amazing summer internship at the IBM Almaden Research Lab. Vasilis Vassalos, for the interesting and constructive discussions on our inconsistency resolution work and for his unforgettable and always funny remarks. Keliang Zhao, for his invaluable help in quickly converting RIDE’s frontend from a paper design to a full-blown user interface that closely followed our sketches on paper.

Thanks go also to all other members and visitors of the database lab for creating such a diverse and great environment both during and after business hours. I will never forget our jokes, our quarterly feasts, our (unsuccessful) attempts at laser tag and our (successful) attempts at teasing each other.

I would also like to thank all my other friends in San Diego with whom we carried out all these seemingly counterproductive activities, that constitute what is called life: eating, drinking, enjoying the beach and in general having fun.

Last but definitely not least, thanks go to my parents, Nikos and Martha, and my brother, Dimitris. Thanks for supporting and encouraging me for all these years until now (and I am sure for many more to come). I hope that one day I can reach your level of patience, politeness and willingness to help other people.

Finally, I would like to acknowledge the co-authors of all my publications that were, in part, the basis of this dissertation. In particular:

Chapter 2 is, in part, a reprint of the research paper published in VLDB 2008 by Yannis Katsis, Alin Deutsch and Yannis Papakonstantinou and contains material published in a demonstration paper in VLDB 2008 by Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou and Keliang Zhao.

Chapter 3 is, in part, a reprint of material that appeared in PODS 2005 and was authored by Alin Deutsch, Yannis Katsis and Yannis Papakonstantinou.

Chapter 4 contains material prepared for publication by Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou and Vasilis Vassalos. Parts of it appear in a
demo paper that is currently under review for ICDE 2010.
VITA

2003 Bachelor of Science in Informatics, Athens University of Economics and Business

2007 Master of Science in Computer Science, University of California, San Diego

2009 Doctor of Philosophy in Computer Science, University of California, San Diego

PUBLICATIONS


ABSTRACT OF THE DISSERTATION

Community-oriented Information Integration

by

Ioannis Katsis

Doctor of Philosophy in Computer Science

University of California, San Diego, 2009

Professor Alin Deutsch, Co-Chair
Professor Yannis Papakonstantinou, Co-Chair

To allow their members to collaboratively maintain the community knowledge, modern online communities need to integrate their members’ structured data into a single community database. Existing integration solutions employed by enterprises are not suited for communities, as they rely on a central authority to carry out the integration tasks and are therefore too costly and not scalable to large numbers of sources.

To solve this problem, we propose the community-oriented integration (CII) paradigm, which removes the need for a central authority by delegating the integration tasks to the individual community members. In this dissertation, we describe how to decentralize two main integration tasks: The registration of sources in the integration system, which becomes the responsibility of each source owner and the resolution of inconsistencies in the collective data, which is delegated to the consumers of the integrated data.

To facilitate this distribution, we introduce a novel architecture and two tools RIDE and Ricolla that assist the community members, who typically lack
the sophistication of the central authority and an overview of the system, in carrying out the source registration and inconsistency resolution tasks, respectively, autonomously.

RIDE models source registrations as sets of Global and Local As View (GLAV) mappings and assists the source owner in creating a registration that balances two competing requirements: Making her data visible to applications that run on top of the community database (by exporting more data) and minimizing the cleaning cost required for publication (by publishing less data). We model these trade-offs as different self-reliance levels, present decidability results and appropriate decision procedures (when existent) and describe an algorithm for interactively guiding the user towards a registration with a particular self-reliance level.

On the inconsistency resolution front, Ricolla models inconsistent data as sets of possible worlds and displays them to the users through a novel data model that summarizes them in an easily understandable and compact form. It also offers a flexible architecture that enables different schemes for the inconsistency resolution, allowing among others users to resolve inconsistencies individually, according to their own opinions, or in collaboration with their peers.
Chapter 1

Introduction

With the widespread adoption of the web, an increasing number of communities, be they scientific, commercial or governmental, strive to build portals that allow their members to collaboratively maintain the community knowledge. However the current technology for data publishing on the web addresses the needs of only the extremes in the spectrum of online communities.

One extreme comprises communities which publish highly structured data into a global database maintained by a central integration authority. For instance, data-driven scientific inquiry needs data generated by multiple scientists and laboratories, which may even cross multiple disciplines. A number of emerging portals, such as GEON [geo] for the geosciences and BIRN [bir] for the biomedical community, aim to provide integrated access to the data of multiple laboratories and scientific communities. Such portals typically rely on traditional information technology employed by enterprises, commonly referred to as Enterprise Information Integration (EII) technology [HAB+05]. Having been developed over a long period of time to meet the demands of enterprises, the EII technology is powerful: It supports complex integration scenarios in which data from different sources are combined in non-trivial ways to form a global database. Moreover, by operating on structured data, it also supports the rich query capabilities typically associated with database management systems. However it also has a major drawback: By relying on a central integration authority to setup and maintain the system, EII systems come at an often prohibitive setup and maintenance cost. Indeed, the con-
struction of portals like BIRN and GEON is still a very large-scale effort, which has a considerable financial cost and takes many years to initiate and accomplish.

The other extreme consists of communities which publish unstructured data (text or multimedia files tagged with attribute-value pairs) with no integration functionality: published data items are not combined, being simply added into the collection. We have recently witnessed the proliferation of such communities with the advent of forums, web blogs, wikis and social networking applications such as del.icio.us [del], flickr [fli] and YouTube [you]. A salient feature contributing to the massive success of these communities is the low setup and maintenance cost due to the decentralized nature which allows new members to join autonomously without assistance from a central authority.

The Problem: Inexpensive Integration of Structured Data for Communities. The above data publishing paradigms leave out the numerous communities whose structured information integration and querying needs preclude the unstructured wiki-style approach and whose limited time or financial budget rules out the costly EII solution. For instance, typical specialized scientific communities lack the resources of GEON- and BIRN-class projects and cannot afford to build infrastructure for the collection, integration and cleaning up of data that pertain to their specialized areas. Graduate students and other researchers end up manually performing information collection, integration, cleaning up, and analysis, at a great productivity cost. The same cost barrier is generally faced by many private, commercial, academic and even governmental communities.

The Solution: Community-oriented Information Integration. This dissertation addresses this need by introducing the community-oriented information integration (CII) paradigm which combines the best of both worlds: As is the case with EII, community-oriented integration systems support integration of structured data with all the associated benefits. Moreover, similarly to social networking applications, this happens at no cost to any central authority. This is achieved by leveraging the power of the community and decentralizing the integra-
tion tasks. These tasks, traditionally carried out in an EII setting by the central integration authority, are in CII systems instead pushed to independent community members.

**Contributions.** CII systems are based on a novel architecture that allows community members to setup and maintain the system in a distributed manner. A major challenge in designing these systems was to enable community members (who typically lack both the sophistication of a central authority and an overview of the integration system) to understand the complex interactions between different sources and users that happen inside an integration system in order to be able to carry the integration tasks autonomously. To this end, the CII architecture contains interactive tools that assist the community members in each of the integration tasks. In particular, we present two interactive tools that empower individual members to autonomously carry out the two main tasks involved in the integration of structured data:

- **Source Registration** (i.e. addition of a new source to the system): In a CII system, this task is delegated to the individual source owners. Each source owner can, utilizing an interactive registration tool, autonomously register her source into the integration system.

- **Inconsistency Resolution** (i.e. resolution of conflicts that arise in the collective data): In a CII system, this task becomes the responsibility of the end users/clients of the system, who query the community data. Clients of the system can, through an interactive inconsistency resolution tool, inspect the inconsistencies in the community database and resolve them either individually or in collaboration with people they trust.

Before describing the community-oriented integration paradigm and these two tools in more detail, we first briefly present the EII, which constitutes the current state of the art in the integration of structured data. ¹

¹Enterprises can nowadays meet their integration needs, apart from EII, also through a variety of other integration tools, such as ETL tools or object-relational mappings. However this work
1.1 Current State of the Art: Enterprise Information Integration

The data management community early on recognized that applications and users need a single point of access to data distributed at multiple sources. For example, a telecommunication equipment vendor may need a portal where, once the customer logs in, the portal uses (i) information from the sales’ department database in order to know which is the product configuration that the customer has bought, (ii) information from the service department’s database to know past complaints of the customer and (iii) the product marketing database in order to suggest to the customer novel products that are compatible with his configuration and are relevant to problems reported in the past.

Academia and research subsequently came up with integration systems and mediators, which provide such a single point of access by exporting a global view that conforms to a global schema against which application developers or end users (also known as clients) formulate their application queries or adhoc client queries, respectively. These systems were later embraced by the industry to form the Enterprise Information Integration technology. At a sufficient level of abstraction, EII systems follow the architecture of Figure 1.1 [Wie92].

Data residing in several local databases (also referred to as source databases or simply sources) are combined to form an integrated database (also known as global database or target database). The global database is usually kept virtual (i.e. it is not materialized per se), similar to the virtual views in database management systems. Subsequently, application developers or clients access the combined information by formulating application queries (or adhoc client queries, respectively) over the global database. However, since the global database is not materialized, these queries cannot be evaluated as usual. Instead, the integration system answers them by appropriately rewriting them to queries against the individual

---

focuses on EII systems, as they have the most well-defined semantics among the existing data integration solutions and as such they form the most widely studied integration solution in the research community. For a comprehensive survey of the integration solutions available to enterprises, the reader is referred to [BH08].
sources, whose answers are collected, combined and returned to the application or end user.

A defining characteristic of EII is that the integration tasks are managed by a central integration authority. This authority decides which data to extract from each source, how to combine them with data from other sources and how to resolve the inconsistencies that might arise in the process.

This makes EII systems very costly to develop and maintain and therefore suitable only for enterprises or very large communities that have the required resources.

### 1.2 Our Proposal: Community-oriented Information Integration

To overcome the cost-ineffectiveness of EII systems, which makes them unsuitable for most communities, we introduce the community-oriented information integration (CII) paradigm. CII leverages the power of the community by pushing the integration tasks traditionally carried out by the central integration authority to the individual community members.

In designing the CII paradigm, we had to decide which tasks need to be decentralized and to whom they should be delegated to. To this end, we inspected the responsibilities of the central authority in an EII system. As explained in
[NBBW06], integrating structured data involves carrying out the following two steps: The first solves the heterogeneity of the sources on the schema level: The administrator decides which attributes to pick from each local/source schema and how to map and reformat them into the global/target schema. We call this process *source registration*. However this does not solve the value discrepancies between the sources: Sources may contain contradicting data, which lead to an inconsistent global database unless they are appropriately cleaned. This is accomplished by the second step, which we refer to as *inconsistency resolution*. The central authority inspects the data conflicts and decides how to resolve them. Please note that these two tasks are often performed in EII systems in a unity as part of a single process. For instance, the decision of resolving conflicts in favor of a particular more authoritative source may be encoded as part of the source registration function that also describes how to change the structure of the source data to conform to the global schema. However we separate them into two distinct tasks, because, as we will explain shortly, in CII it makes sense to delegate each task to a different type of community members.  

**Push the source registration to the source owners.** In a decentralized setting the source registration should be accomplished by the owner of each source, as she is both most knowledgeable about the contents of her source and responsible for deciding which data from her source should be exposed to the users of the integration system. In contrast, she is not suited for removing the inconsistencies from the data, as she is likely to be biased and resolve them in favor of the data originating from her own source.

**Push the inconsistency resolution to the clients.** The inconsistencies should instead be resolved by the clients of the integration system, who query the

---

2As described in [NBBW06], the integration process also consists of a third step, known as entity resolution or duplicate detection: The task of deciding when two tuples from different sources refer to the same entity and should therefore be merged into a single tuple in the global database. In this work, we assume that tuples referring to the same real-world objects share the same primary key values and therefore entity resolution is not required. However entity resolution mechanisms can be incorporated into the CII paradigm as we discuss as part of our future work in Section 4.10.2.
community data and are therefore motivated to resolve the inconsistencies in order to improve the quality of the query results. The need to push the inconsistency resolution to individual end users is even more pronounced in communities where inconsistencies do not correspond to mistakes that can be resolved in a unique, globally agreed way but instead to different opinions across members. For instance, in science it is often the case that different labs disagree in their findings. Moreover the community cannot agree as a whole with one of the contradicting opinions, as they correspond to theories that cannot yet be verified. In these settings, an EII system is not only cost-ineffective, but also incompatible with the inner workings of the community: It essentially forces every community member - even if she has a different opinion - to accept as the truth a single opinion (which was arbitrarily chosen by the global administrator). Instead these communities need a system, where the inconsistency resolution is performed by each individual client, who can resolve inconsistencies independently of others according to her own beliefs.

**Challenges.** Hence, CII systems push the source registration to the individual source owners and the inconsistency resolution to the individual clients. However transforming an EII system into an integration system where the integration tasks are decentralized, is a challenging and non-trivial process for two reasons. First, EII systems are designed with a central authority in mind and as such their architecture inherently does not allow certain tasks to be performed independently by multiple persons. For instance, as we will explain later, most EII systems employ a source registration language that only allows to register all sources *at once* by describing *in a single step* how data from different sources are combined and mapped into the global schema. This precludes the independent source registration, where the owner of a new source should be able to add her source to the integration system without being required to either inspect or modify the registration of other sources. Therefore, CII systems need an entirely *new architecture* and different internals from EII systems. Moreover, converting an EII system to a decentralized system is also hard for another reason: Individual
community members, who take over the tasks previously performed by the central integration authority, typically lack the sophistication of the latter and they do not have (and have no hope to acquire) a global overview of the system. Therefore, source owners cannot grasp the ramifications of how their data interacts with the application queries and other sources, and clients cannot understand which are the inconsistencies and where they originate from to be able to resolve them. Hence, community members are not able to carry out the source registration and inconsistency resolution tasks, unless they get appropriate assistance. To this end, the proposed CII system contains two interactive tools that assist the community members in each of the two integration tasks: the source registration and the inconsistency resolution.

**CII Architecture.** Figure 1.2 shows the resulting architecture of a CII system. Each individual source owner registers her source autonomously utilizing the source registration tool. In parallel, clients access the data of the integration system and resolve inconsistencies either individually or in collaboration with each other by employing the inconsistency resolution tool.

We detail next the issues community members need assistance with for both source registration and inconsistency resolution, and the tools we propose for providing this assistance.
1.2.1 Source Registration

To assist a source owner in the registration of her source, we must first comprehend her incentives behind this action. A source owner registering a new source does so motivated by the desire for her data to be visible to certain applications heavily used by the community. For instance, a member of a community of used Jazz LP album seekers wants to know whether her ads are visible to the queries issued by the brokerage application against the global schema.

An overkill (if at all possible) way to ensure this is to force the source owner to map some data into every attribute of the global schema relevant to the application query. This approach may be simply impossible because a source owner may not possess data for some parts of the community schema. For instance, the application query may retrieve only LPs with good reviews but our source owner may have no reviews to offer. The visibility of the new member’s albums thus depends on the existence of other sources. For instance her LP ads will be returned only if some other source provides pertinent reviews which “join” with her ads. Since the owner has no overview of all existing sources in the system, she needs a registration tool that assists her in determining and, if possible, increasing the level of visibility of her data.

More importantly, even when a source owner is in a position to map data into all parts of the community schema which are relevant to the application query, it may be economically unwise to provide values for all the attributes. In particular, global schemas developed by communities are often very expansive and detailed\(^3\). It may be an overkill to clean up the data in order to populate all possible attributes with the appropriate format. For example, consider a biologist who has annotations about possible physiological aspects associated with a gene. Medical ontologies have codes for many terms. The biologist may not want to painfully find them and parse the annotations in order to specify all attributes (including trivial ones) of the global schema. Instead, he may prefer to clean up only attributes that are pertinent to the existing application queries and leave the rest in

\(^3\)The reader is referred to the DTDs and XML Schemas of OASIS \url{www.oasis-open.org} for examples.
RIDE: A tool for community-oriented source registration. To help source owners negotiate this complex trade-off between increasing visibility of their data to existing application queries and minimizing the cost for cleaning and reformatting their data, we propose RIDE (Registration guIDE); a visual interactive registration tool. The first step in developing such a tool was to characterize the self-reliance levels of a given source registration to a given query, each of them corresponding to a different trade-off between visibility and cleaning cost. RIDE allows the source owner to select the desired self-reliance level and explains to her which minimal amount of data she must register into the community schema to achieve it.

Figure 1.3 shows a screenshot from a user interaction with RIDE. Source registrations are modeled as sets of schema mappings and an owner can create the registration of her source through a series of visual actions, such as drawing lines and entering constants on the interface (which correspond to formally defined changes to the underlying schema mappings). At each interaction step, RIDE utilizes a guidance algorithm that computes the possible ways in which she can extend her current registration to one with the desired self-reliance level and presents them to the user through appropriate coloring of the interface and/or specially formulated questions about the contents of her source.
The system is based on the following contributions: a) Formal definitions of a set of self-reliance levels of registrations, which are defined as sets of GLAV (Global and Local As View) schema mappings, b) efficient procedures both for deciding the self-reliance levels and for guiding the user to an extension of her current registration with the selected self-reliance level, c) a prototype implementation of the tool and d) an experimental evaluation proving the applicability of the tool in large communities by showing that the proposed algorithms scale with the number of registered sources.

1.2.2 Inconsistency Resolution

The global database inevitably ends up containing inconsistent data, as its data originate from different sources, which commonly contain mistakes or data corresponding to different opinions. Current integration systems based on EII force the administrator to resolve the inconsistencies before the data become available to the end users in the form of the global database. However this is unreasonable to expect in a CII scenario, where each client cannot (and should not) be expected to resolve the entire database before being able to query the data. Instead clients need a system that tolerates inconsistencies and is fully functional (allowing them for instance to query the data) in the presence of conflicting data. Moreover such a system should allow them to inspect the inconsistencies and resolve them either independently or in collaboration with other users (to support scenarios where the inconsistencies correspond to differing opinions). In particular, clients in an CII need a system that contains provisions for the following tasks:

- *Inspect the inconsistencies.* To be able to utilize the data, a client should be provided with ways to easily identify the existing inconsistencies and the possible ways in which they can be resolved.

- *Resolve the inconsistencies in an “as-you go” fashion.* The system should not mandate the resolution of all inconsistencies before the data can be used. Instead it should support a natural “as-you-go” inconsistency resolution, in which users interleave inconsistency resolution steps with regular data
management tasks, such as querying data.

- **Resolve the inconsistencies in different ways, depending on the application scenario.** To capture a wide variety of inconsistency resolution scenarios, the system should not enforce a particular resolution policy. Instead clients should be allowed to create their own policies on how data can be resolved. The supported policies should among others include policies allowing the users to resolve data individually (to support settings where different users have different opinions) as well as policies allowing the clients to resolve data collaboratively (to support settings where peers proficient in part of the community data, help their colleagues by resolving the inconsistencies pertaining to their area of expertise for the entire community to see).

**Ricolla: A system for community-oriented inconsistency resolution.** As a first step towards creating an inconsistency-resolution tool for CII systems, we propose Ricolla; a wiki-inspired online database system that, by treating inconsistencies as first-class citizens, supports a natural workflow for the management of conflicting data. The system captures inconsistencies (so that community members can easily inspect them) and remains fully functional in their presence, thus enabling inconsistency resolution in an “as-you-go” fashion. Moreover it supports several schemes for the resolution of inconsistencies, allowing among others users to collaboratively resolve certain conflicts while disagreeing on others.

The system is based on the following technical contributions: a) a novel architecture that tolerates inconsistency, allowing data query and update, while
enabling and aiding inconsistency resolution by community members, b) a data model and corresponding interface for explaining the inconsistencies to the users. These capture inconsistencies as sets of possible worlds and present them in a summarized form utilizing colors and nesting to indicate the correlations between data values (see Figure 1.4 for a sample screenshot of Ricolla), c) a set of resolution actions that allow each user to resolve individual data inconsistencies, d) a resolution policy language for summarizing a set of resolution actions based on high-level criteria (e.g. give priority to data according to their provenance or based on timestamps) and e) a set of algorithms for implementing the system on top of a RDBMS.

Please note that Ricolla is currently applicable to a particular case of the integration scenario depicted in Figure 1.1. In its current incarnation it provides support for inconsistency resolution in community scenarios where individual members insert pieces of data manually one-by-one (in contrast to the bulk uploads of data from local databases, which is commonly the case in integration systems and the basis for the architecture of CII). The decision to factor out the bulk insertion component was made for the following two reasons: First, it allowed us to focus on the core problem of inconsistency resolution independently of the issues cause by the schema mappings. Second, the resulting system - Ricolla - is (even without bulk insertions) already applicable to a large number of applications that allow community members to share their data by entering it manually by hand. This class of applications, referred to as online databases, has been seeing a tremendous boost during the last few years, with multiple products being offered, such as Dabble DB [dab], blist [bli], Quickbase [qui], Freebase [fre] and Google Fusion Tables [fus]. The extension of Ricolla to bulk insertion of data and its seamless integration with RIDE, which is left as future work, is discussed in Section 4.10.

1.3 Outline

This dissertation is structured as follows: Chapters 2 and 3 discuss source registration and Chapter 4 presents our work on inconsistency resolution. In par-
ticular, Chapter 2 shows how to decentralize the source registration and describes RIDE. Chapter 3 extends the discussion of the previous chapter by presenting alternative definitions of the self-reliance levels together with theoretical results on their decidability in the presence of different classes of constraints over the global schema. Chapter 4 corresponds to the inconsistency resolution component of our work and presents Ricolla. Finally, Chapter 5 concludes the dissertation and discusses future work.
Chapter 2

Source Registration

This chapter describes RIDE; a interactive tool that decentralizes the source registration process by allowing each source owner to autonomously register her source in an integration system. The chapter is structured as follows: We fist present some necessary background and an overview of the system in Sections 2.1 and 2.2, respectively. Then we start the detailed description of the system by first presenting RIDE informally, describing our running example in Section 2.3, traditional schema mapping GUIs (which form the basis for RIDE’s user interface) in Section 2.4, and a sample interaction highlighting RIDE’s functionality in Section 2.5. Section 2.6 formally defines the tool. The algorithms underlying RIDE and their experimental evaluation are described in Section 2.7. We discuss related work in Section 2.8, future work in Section 2.9 and conclude in Section 2.10.

2.1 Background

In this chapter we focus on assisting individual members to autonomously join a CII system by registering their data into the integration system. Compared to the well-studied source registration in EII systems (which is performed by a central integration authority), autonomous source registration creates new challenges caused by each community member’s lack of a global overview on how her data interacts with the application queries of the community and the data from other sources. How can the source owner maximize the visibility of her data to existing
applications, while minimizing the clean-up and reformatting cost associated with publishing? Does the source owner’s data contradict (or could it contradict in the future) the data of other sources? Previous work on data integration did not consider these questions, since the central authority’s global overview made them non-issues. Autonomous registration on the other hand is impossible if we do not answer them. We detail next the issues community members need assistance with.

**Contribution to application query results.** A source owner registering a new source desires her data to be visible to relevant client applications that issue queries against the community’s global schema, which we will call *target schema*, in keeping with the terminology of IBM’s Clio system [YP04, FKMP05]. For example, a book retailer joining a community of bibliophiles wants her book ads to be visible to queries issued by a popular brokerage application.

An overkill (if at all possible) way to ensure visibility is to force the source owner to map some data into every attribute of the community’s target schema relevant to the application query. For instance, the application query may retrieve only books with high ratings in their reviews. If the retailer can publish reviews along with its ads, her registration will be “self-sufficient”, in the sense that her books will be visible to the application query regardless of the contents of other sources registered in the community.

This solution may be simply impossible because a source owner may not possess data for some parts of the community schema. Our retailer may have no reviews to offer, in which case the visibility of her books depends on the existence of some other source providing pertinent reviews that “join” with her ads. In this case, the retailer would like to know that her registration is no longer self-sufficient, being instead complementary to that of the review source.

**Trading off self-reliance for cleaning cost savings.** Even when a source owner is in a position to map data into all parts of the community schema that are relevant to the application query, it may be economically unwise to do so, due to the prohibitive clean-up and reformatting cost. In such cases, the source owner may willingly give up self-sufficiency, settling for a “complementary” registration that relies on other, trusted sources. In this case, the registration
tool would best serve the owner by labeling the publishing of appropriate data attributes as optional and identifying the partner source(s) that could provide them instead. Looking at the options, the owner can then decide herself which trade-off between self-reliance and cleaning cost savings she wants to take.

In the running example, assume that the retailer collects third-party reviews in the form of text blurbs. Cleaning them up for publication (spell-checking, language censorship, etc.) and formatting them to extract certain measures required by the community’s target schema (such as star ratings and a representative quote) is an expensive process requiring human involvement. If the registration tool notifies the retailer of another trusted source that provides reviews, she may choose to rely on this source and save the effort.

**Inconsistency avoidance.** To reach their full potential, community-oriented integration systems should enable the combination of data provided by distinct sources into a single target tuple, using standard integrity constraints. For instance, the book dimensions (provided by the publisher’s source) are associated with the book’s price (given by the retailer) by virtue of both data items referring to the same ISBN declared as a key on the target schema. Target constraints may however lead to inconsistency, for instance if publisher and retailer list different authors for the same ISBN. Since the publisher and retailer do not know each other’s registrations, inconsistencies are even more likely than in centralized integration scenarios. It is therefore imperative for a registration tool to identify registrations leading to inconsistency and issue warnings.

### 2.2 Overview of RIDE

To facilitate autonomous source registration, we propose Registration guIDE (RIDE), a visual tool that extends the classic schema mapping interface (as encountered in IBM Clio [YP04, FKMP05], MS BizTalk Server [MSB] and Stylus Studio [Sty]) with a suggestion component that guides the source owner in the registration of her source. The suggestions assist the source owner to negotiate the trade-off between two competing requirements: maximizing self-reliance for
making her data visible to existing application queries, versus minimizing the data cleaning and reformatting cost. In addition, RIDE helps the owner avoid inconsistency of her data with respect to data in other sources.

The resulting architecture of a community-oriented integration system enabled by RIDE is shown in Figure 2.1. In such systems the community initiator starts by designing the target schema. Note that although the initiator might be a consortium agreeing on a common schema this is not necessary for starting a community. Most commonly we envision the emergence of ad hoc communities whose initiator (possibly an individual) decides the schema without seeking source owner approval. The community will attract more and more members as it gains in popularity in the same way that online communities like blogs grow. ¹ After the initiation of the community, application developers register within the system the queries (over the target schema) that their applications will issue during operation. To register a new source into the community, its owner chooses an application query to which she wants the source to contribute, as well as the desired self-reliance level (from a pre-defined list of options detailed below). She then initiates a registration process with RIDE. ²

RIDE’s visual interface allows owners to perform such actions as drawing arrows between their source schema attributes and the target schema attributes.

---

¹Ad hoc communities may also evolve. For instance, a community may have its target schema changed or it might be coalesced with another ad hoc community on the same topic. For a discussion on how a community-oriented integration system can support such evolution aspects the reader is referred to our discussion on extensions and future work in Section 2.9.

²In this work, we do not address the run-time aspects of the integration system, such as the problem of answering queries over the target schema once the registrations are given.
they want to provide, and also depicting selection and join conditions to restrict the publishing. RIDE interactively suggests what target schema attributes to provide and which selection conditions or join conditions to employ in order to reach the desired self-reliance level. The list of suggestions adapts to the owner’s action at each interaction step, to include only attributes that are essential and that the owner is willing to provide.

There are many consistent registrations that feature the same self-reliance level. The source owner may prefer some of them over others, as she trades off cleaning cost savings (by restricting the published data to only the minimum relevant to a query) for generality of the registration (by publishing more than needed to contribute to a query, in order to contribute to others as well). RIDE assists the source owner by laying out the available options.

2.2.1 Contributions

Inconsistency and self-reliance levels. To formalize the provided functionality, we characterize the self-reliance levels of a given source registration to a given query, as detailed in Section 2.6.3. Higher levels require publishing more data fields, which yields less reliance on what other sources provide, but in exchange may involve more cleaning effort. With respect to an application query $Q$ formulated against the target schema, a registration $R$ can be (in decreasing order of self-reliance):

- **self sufficient** if it contributes answers to $Q$ even if all other sources leave the community;
- **complementary** if it contributes answers to $Q$, but only in cooperation with the registrations of some other sources from the community;
- **unusable** if it is none of the above.

We also formalize two notions of inconsistency, namely

- **potential** inconsistency, which may occur for some contents of the source databases, and
- **definite** inconsistency, which will occur for all source databases.
Potential inconsistency is more of a conservative property checked at registration time, since whether it will actually occur at run-time will depend on the data in the source databases. Definite inconsistency on the other hand is a serious problem, since it will always appear at run-time regardless of the source data. Although definite inconsistencies would not exist in an ideal world, human errors in the registration process may introduce them. RIDE detects them and can either reject the registration or simply issue a warning.

**Guidance algorithms.** We implement algorithms that at each interaction step, (a) check inconsistency, (b) find the current self-reliance level and (c) compute suggestions on how to extend the registration to one with the desired self-reliance. We report on our experimental evaluation which shows the response times of these algorithms to be well within the needs of an interactive visual tool.

**Guaranteed inconsistency avoidance.** The tool guarantees that, by following its suggestions, the desired self-reliance level can be reached without inconsistency. If the owner chooses not to follow the tool’s suggestions and creates an inconsistency, RIDE explains how this inconsistency can come about.

**Data-independent guarantees.** The self-reliance level of a registration can hold with respect to all possible instantiations of the source databases, or only to the current instance of the sources. We call these the data-independent, respectively data-dependent flavors of self-reliance guarantees. Both flavors come with their own benefits and drawbacks: Data-dependent guarantees need to be re-evaluated upon updates of the underlying data sources and hence the source owner will be continuously and annoyingly alerted for changes of the guarantees pertaining to her registration. Data-independent guarantees may be too strong, in the sense that they may alert for potential violations that are due to source instances where common sense about the domain may indicate that these instances are impossible or improbable to happen.

In this work, we aim for a balance between data dependence and independence. To this end, we consider guarantees that hold over a restricted class of source databases. As long as source updates leave the sources within the same class, consistency and self-reliance levels are preserved and need not be re-checked.
The classes are specified to consist of those databases that satisfy integrity constraints and assertions. Integrity constraints are declared by the source owner, while assertions are constraints generated by RIDE and presented as questions to the owner, who may confirm or refute them. The self-reliance level and consistency of a registration are thus guaranteed as long as the integrity constraints and assertions hold. Efficient checking that an update violates integrity constraints and assertions has been addressed extensively in related work and is beyond the scope of our work (see related work in Section 2.8).

2.3 Running Example

We demonstrate RIDE using as our running example the creation of “Bibliophilia”; an application for the bibliophiles’ community that integrates book information from several sources. The community’s target schema $S$, shown in Figure 2.2b consists of two relations Book and Book_Price, shown in italics. Relation Book contains general information about a book, while Book_Price stores the regular and the discounted price (i.e., price for “Bibliophilia” members) at which sellers provide the book. Underlined attributes correspond to (composite) primary keys.
Owners of sources with book data that want to make these data available to Bibliophelia’s client applications can use RIDE to register their sources within the system. In the following we use two sources; the bookstore *Barnes & Noble (B&N)* and the publisher *Prentice Hall (PH)*, with schemas shown in Figure 2.2a. The Barnes & Noble database stores ISBN, title and author of hardcover books in the *Hardcovers* relation and prices of books for different classes of customers (i.e. non-members, gold members etc.) in the *Price* relation. Similarly, relation *Book_Info* of Prentice Hall contains the ISBN, title, binding and suggested retail price of books.

Each source owner initiates the interaction with RIDE by selecting an application query to which she wants to contribute. In our example this is the query retrieving title and format of books by Ullman together with the regular and discounted price at which they are sold. This query is shown in Figure 2.3. Note that in general application queries can be parameterized (e.g. the author name could be a parameter). However a non-parameterized query allows us to showcase all features of RIDE. In the presence of parameters in the application query, RIDE still provides suggestions, but fewer in number due to lack of knowledge of the parameter’s value. We discuss parameterized queries in Section 2.5.3.

### 2.4 Mapping Interfaces

RIDE’s front end resembles graphical interfaces of schema mapping tools, such as IBM Clio [YP04, FKMP05], MS BizTalk Server [MSB] and Stylus Studio [Sty]. These allow users to create mappings between two schemas by drawing lines between their respective attributes. Similarly, RIDE enables source owners to register their sources by creating one or more mappings between their source schema and the target schema solely through visual actions. Figure 2.4 depicts a mapping of the B&N source created through RIDE.

Owners specify mappings via the following actions:

Drawing *projection arrows* from a source attribute to a target attribute, to specify where the latter gets its value from. For example, in Figure 2.4, the
price for gold members in the B&N database is exported as the discounted price in Bibliophilia’s database.

Entering (source / target) selection conditions next to attributes. A source selection condition restricts the exported source data to those satisfying the condition. For example, the condition “class = gold members” in Figure 2.4 limits the exported prices to only those for gold members. A target selection condition allows the source owner to enter information in the target database that is not stored explicitly in the source database. For instance, B&N’s owner in Figure 2.4 specified through target selection conditions that her books are hardcovers and that her bookstore’s name is “B&N”.

Drawing (source / target) join lines between pairs of source / target attributes. Join lines have similar semantics to selection conditions with the only difference that they represent equalities between two attributes instead of equalities between an attribute and a constant. For instance, B&N’s source owner in Figure 2.4 employs a source join line between the ISBNs to export only pairs of book and price tuples that join on the ISBN. Additionally, she uses a target join line to declare that B&N sells books to non-members at the suggested retail price, regardless of what this price may be.

The owner can always extend her registration with additional mappings. Each mapping appears as a vertical tab on the interface.

We formalize the semantics of mappings in Section 2.6.1.
2.5 RIDE Interaction

In this section we informally present both the suggestion and the inconsistency component of RIDE via sample interaction sessions. The formal definition of the involved concepts (such as self-reliance levels and inconsistency) can be found in Section 2.6.

We first provide key principles and characteristics of the RIDE interface and then describe the suggestions it provides, escalating to suggestions that are hard to discover without RIDE’s assistance.

Starting from an initially member-less community, we first show how the B&N source’s owner interacts with the system to obtain a self-sufficient registration w.r.t. the query of Figure 2.3. Then, assuming B&N joined the system, we present an interaction session led by the owner of Prentice Hall, who wants to create a complementary registration w.r.t. the application query and B&N’s registration. Figures 2.5 and 2.6 depict the respective screenshots.

2.5.1 Suggestion Component

Using RIDE the source owner can achieve the following:

**Focusing on attribute subsets.** To contribute to a query $Q$ the source has to provide a *subset* of the target attributes that are *required* by $Q$; i.e. attributes that are selected, projected or joined by $Q$. In general, the source owner has several options between different subsets of required attributes that she can provide to gain the desired self-reliance level; she could provide all attributes for a self-sufficient registration, or various attribute subsets to achieve complementarity with various sources.

*For example, if the integration system already contained two sources, one providing Ullman book information and the other Ullman book prices, then the new source could become complementary w.r.t. the query of Figure 2.3 by either providing book prices exported by the first source or book information sold by the second.*

Without assistance the task of finding all subsets of required attributes that
lead to the desired self-reliance level is infeasible. It requires understanding the registrations of all sources and figuring out how data from existing sources can be merged with each other and complemented with data from the current source to form an answer to the query. To assist the owner, RIDE computes all such subsets and displays them in the gray pane to the right. Each subset is depicted as a vertical line pointing to the attributes in the set. Required attributes are marked in bold face. However the owner can do more than simply see all available options. By clicking on the subset of attributes she is willing to provide, she can instruct RIDE to generate only suggestions pertaining to this set, avoiding thus suggestions of no interest to her. Finally, apart from guiding the search, she can also use the right pane to get a quick overview of which required attributes have yet to be provided.

In the running example, due to the small number of sources, there is only one such subset. The right panel shows which required attributes have to be provided, as seen in all snapshots of Figure 2.5.

Once the owner selects a subset of required attributes, she can see the different possible ways to provide a particular attribute by clicking on it. RIDE marks the selected attribute with a green flag to its left and shows the suggestions by shading interface components. Suggestions are replicated in text on the bottom status bar.

Directly providing attributes. The easiest way to provide a required attribute is by directly mapping to it values from some source attribute (through a projection arrow) or assigning to it some constant value (i.e. entering a target selection condition). RIDE shows these suggestions by shading the projection arrow box and selection condition box next to the attribute, respectively. \footnote{Please note that currently RIDE only suggests the target of arrows (i.e. which target attribute to provide through an arrow) but not their origin (i.e. where to map it from) as it is not aware of the semantics of the source and target schemas. However it could be coupled with a matching tool to also suggest arrow sources.}

For example, Snapshot 1.1 shows that to create a self-sufficient registration
Figure 2.5: Sample interaction for the B & N registration
w.r.t. the query of Figure 2.3, B&N has to provide Book. ISBN either through a projection arrow or a selection condition.

Trading off cleaning cost savings versus generality. Directly providing an attribute through a projection arrow does not always suffice to acquire the desired degree of self-reliance to a query $Q$: the source may only contribute to $Q$ if it contains tuples with specific values asked by the query (books by Ullman in our running example). In this case, RIDE offers the source owner two options, each achieving a different trade-off between cleaning cost and generality of the registration mapping.

Source selections. If the owner wishes to minimize cleanup cost, she can restrict the exported tuples to only those with the particular value asked by $Q$. RIDE will suggest the corresponding source selection option.

Intra-source assertions. However, if for the sake of contributing to several queries the owner prefers to export more tuples than those relevant to $Q$, she may choose to not include the selection condition in her mapping. In that case, RIDE will ask her if she believes that the exported tuples will always include at least one tuple relevant to $Q$. If she answers positively, RIDE records the answer and takes it into account when generating subsequent suggestions.

In designing this dialog, we chose a solution according to which RIDE’s questions are expressed in terms of the source schema (which the owner understands best) and have a standard graphical representation: RIDE presents the owner with boolean queries over her own source schema. Such queries are called assertions, and are displayed by RIDE in dialog boxes using the classical visual paradigm developed for Query-By-Example interfaces such as the query builders of MS Access and MS SQL Server [ms-].

For example, consider Snapshot 1.2 showing the suggestions for Book.author after the source owner manually mapped source attributes into ISBN, title and author and entered a constant into format. RIDE uses the information that the query is asking for books by Ullman and notifies the user that author is not yet provided. She can provide it either by limiting the exported hardcovers only to those by Ull-
man (through a source selection condition) or by accepting the assertion (generated by RIDE) that her source contains at least one Ullman hardcover.

The source owner faces a similar trade-off when the query filters its input tuples using joins instead of selections. Again, RIDE generates two kinds of suggestions: including the join for aggressive minimization of cleanup cost, versus dropping it but asserting the existence of tuples that satisfy the join condition.

For example, assume that B&N’s owner accepts the assertion of Snapshot 1.2 and continues by entering selection conditions and projection arrows. When she draws a projection arrow into \texttt{Book.Price.ISBN}, RIDE knowing that the query asks for books and corresponding prices (a) suggests a source join and (b) shows via an assertion (in Snapshot 1.3) that, in order for the \texttt{Price} tuples to contribute to the query, the source database needs to contain an Ullman book in table \texttt{Hardcovers} that joins on \texttt{ISBN} with a \texttt{Price} tuple. Notice how the assertion from Snapshot 1.2 is used in generating the assertion of Snapshot 1.3 to indicate that, to contribute to the query, the join must involve Ullman books.

The expressiveness of the registration mappings and the intricate ways in which data across sources can interact with each other via the target constraints give rise to subtle ways of contributing to the application query, which are hard to discover by an unassisted owner lacking an overview over the other registrations.

Data merging. Data merging allows the source owner to minimize the cleanup cost (at the expense of self-reliance) by providing only part of the required attributes and “borrowing” the remaining part from other sources. This becomes possible whenever both the owner’s source and the complementary source export partially specified tuples into the same target table, sharing the key value.

For example, recalling that the PH source schema does not carry author information, no registration of the PH source can become self-sufficient for the query in our running example. However, the author value will be automatically “borrowed” from B&N for all PH and B&N books sharing the same \texttt{ISBN} value.
RIDE will in this case suggest to the PH owner to provide the `Book.ISBN` attribute on the way to a registration complementary to B&N.

**Indirectly providing attributes.** So far we have seen cases where the source directly provides a required target attribute through a projection arrow or selection condition. However, a source owner may be able to provide an attribute value *indirectly* by operating on a *different* target attribute. RIDE identifies such non-obvious opportunities and makes the appropriate suggestions. The following example illustrates a case in which an attribute can be provided indirectly by the PH source, while the others are borrowed from B&N to achieve complementarity.

Assume that B&N’s owner accepted the assertion of Snapshot 1.3 and subsequently extended her registration to the one shown in Snapshot 1.4. Recall that this is the registration we saw in Figure 2.4, which does not provide B&N’s regular price for books, stating instead that it equals the suggested retail price. Consider now the interaction step in the registration of PH, shown in Snapshot 2.1 of Figure 2.6. Based on the equality of prices expressed by B&N’s mapping, RIDE shows that PH can become complementary w.r.t. the query of Figure 2.3 if it merges its data with the B&N data by providing the regular price either directly or indirectly by instead providing attribute `Book.sug.retail`, as well as the key `Book.ISBN` (needed for merging). The label “B&N” next to `sug.retail` shows that the indirect provision is facilitated through B&N’s mapping.

**Inter-source assertions: supporting data merging.** While data merging requires that both sources provide values for the key attributes, this is not sufficient. The sources must also provide tuples sharing the key value. Upon identifying data merging opportunities, RIDE therefore asks the owner (via an assertion dialog box), whether her source has tuples that join with those of the other source. When designing inter-source assertions, the challenge was to pose such questions in terms of the only schemas a source owner may be expected to be familiar with: her own source schema and the target schema. As a result, the part of the assertion referring to the other source schema is shown in terms of the
other source’s contribution to the target schema.

Example: Assume that the PH’s owner follows RIDE’s suggestions in Snapshot 2.1 and provides the ISBN and suggested retail prices of books. In order to provide the regular price of some book to the query, she has to make sure that she exports at least one of Ullman’s hardcover books sold by B&N. Therefore RIDE asks her if she wants to make the assertion shown in Snapshot 2.2.

2.5.2 Inconsistency Component

After each user action, RIDE checks the registration for inconsistency. If a potential (respectively, definite) inconsistency is detected, it marks with a “!” (resp. “X”) the attribute for which two conflicting values may be provided (respectively in the case of definite inconsistency, are provided) and explains graphically the root of the inconsistency in a gray box at the bottom of the screen. The following two examples illustrate cases of potential and definite inconsistency, respectively, and RIDE’s reaction.

Continuing our running example, assume that PH’s owner accepts the assertion on Snapshot 2.2 and thus creates a complementary registration. If subsequently she extends the mapping to also export book titles, this creates a potential inconsistency, since B&N and Prentice Hall could provide different titles for the same book. The gray box in Snapshot 2.3 visually depicts this conflict.

While potential inconsistency is quite common, definite inconsistency usually results from human error as the next example shows.

Example: Assume that at some point in the future PH decides to store only paperbacks in its relation Book_Info and provides this information to the integration system through a target selection condition on format, resulting in the registration shown in Snapshot 2.4. In this case the system becomes definitely inconsistent, since PH provides a book in common with B&N (due to the accepted assertion of Snapshot 2.2) and therefore this book has to be both paperback and hardcover. RIDE notifies the user by explaining the inconsistency as shown at the bottom of the Snapshot.
Figure 2.6: Sample interaction for the Prentice Hall registration
2.5.3 Dealing with Parameterized Queries

Our running example employed a single query to showcase the entire set of suggestions that RIDE can generate. However RIDE continues to make non-trivial suggestions even for parameterized queries, which are commonly used by applications running on top of databases. In particular, if we replace a selection with a constant in a query with a parameter, RIDE’s suggestions for the non-parameterized attributes stay the same. The only suggestions that cease to exist are selection suggestions for the parameterized attributes as those would defeat the generic purpose of the parameter. For instance if in our running example author was a parameter, RIDE would make the same suggestions apart from the ones shown in Snapshot 1.2 (while removing ‘Ullman’ from all others).

For simplicity’s sake in the rest of this chapter we talk about non-parameterized queries only. However all definitions and algorithms can be straightforwardly extended to parameterized queries.

2.5.4 RIDE’s Properties

Our design of RIDE was guided by the following desiderata:

- **Soundness of suggestions:** RIDE only makes suggestions that are guaranteed to lead to registrations of the desired self-reliance.

- **Suggestions relevant to owner’s focus:** RIDE only makes suggestions that are relevant to the owner by allowing her to guide the search in several ways: First, she chooses a subset of attributes to provide and thus avoids seeing suggestions on attributes that she cannot or is unwilling to provide. Second, she specifies source constraints or accepts assertions (proposed pro-actively by RIDE) about her data. Both of these restrict the structure of the source database and are exploited by RIDE to skip suggestions and warnings if they do not apply to data satisfying the restrictions.

- **Adaptive response to user’s actions:** RIDE does not pre-compute all suggestions beforehand but instead recomputes them adaptively after each user action. It does so even when the user ignores its suggestions and carries out a non-suggested action instead.
2.6 Formal Specifications

In this section we describe the framework of community-oriented integration and provide formal definitions for the RIDE concepts described in Section 2.5. These definitions will be utilized in the description of the algorithms in Section 2.7.

2.6.1 Community-oriented Integration

Community-oriented integration systems should allow a new source to register without having to modify the registrations of other sources. This requirement affects our choice of registration formalism. In particular, it precludes the use of the Global As View (GAV) approach to data integration, employed by commercial technology, because in GAV each target relation is described as a view over all sources, which has to be revised whenever a new source joins. Instead we have to choose between the other two main integration approaches, namely Local As View (LAV) or Global-Local-As-View (GLAV). We use GLAV [FLM99, Hal00, Ull00, YP04] for its expressiveness. GLAV, which generalizes both GAV and LAV, allows registrations that gather data from multiple source relations into a single target tuple; a feature not supported by LAV. For a thorough discussion on the different approaches in data integration, see [Hal00, Len02].

Source and Target Schemas and corresponding Constraints. A community-oriented integration system integrates a set of source (local) databases with source schemas $S_1, S_2, \ldots, S_n$ through a virtual target (global) database over target schema $G$. Both the source and target schemas are relational and may include integrity constraints, called source and target constraints, respectively.

Owners formulate constraints from the class of embedded dependencies, which are expressive enough to capture many common integrity constraints, such as primary keys (PKs) and foreign keys (FKs), inclusion, multi-valued, join dependencies and beyond [AHV95]. In the following, $\Delta_{S_i}$ denotes the set of all source constraints over source schema $S_i$. Similarly $\Delta_G$ represents the set of target constraints. A database instance $DB_i$ over schema $S_i$ satisfies the set of constraints $\Delta_{S_i}$, denoted as $DB_i \models \Delta_{S_i}$ if it satisfies all constraints in the set.
Registrations and Mapping Constraints. The correspondence between a source schema $S_i$ and the schema $G$ is defined through the source registration $R_i$. According to GLAV, a source registration $R_i$ is a set of mapping constraints (also called mappings). Each mapping constraint is of the form $U \subseteq V$, where $U, V$ are conjunctive queries with equalities ($CQ^=$). These capture Select Project Join SQL queries augmented by a WHERE clause consisting of equality conditions between attributes and constants. $U, V$ are formulated against the source schema $S_i$ and the target schema $G$, respectively. Intuitively, these constraints specify that, given a source database $DB_i$ and a target database $G$, the source data identified by running $U$ over $DB_i$, is visible among the target data identified by running $V$ over $G$: $U(DB_i) \subseteq V(G)$. We say then that the pair $(DB_i, G)$ satisfies the mapping constraint, denoted as $(DB_i, G) \models (U \subseteq V)$. Note that there are no containment statements in the opposite direction, because a local source owner cannot know what information the other sources contribute and therefore cannot presume to contribute all target data. This is consistent with the widely accepted open world assumption [Hal00, Len02].

Every mapping visually specified in RIDE is interpreted as a mapping constraint of the above form. For each projection arrow between source attribute $a$ and target attribute $b$, attributes $a$ and $b$ appear in the projection lists of $U$ and $V$, respectively and in the same position. Moreover each source (target) join corresponds to a join in $U$ (respectively $V$) and each source (target) selection condition corresponds to an equality condition with that constant in $U$ ($V$). For instance, B&N’s mapping shown in Figure 2.4 corresponds to the mapping constraint $U \subseteq V$ with $U, V$ given below:

$U(I_1, T, A, I_2, P) : \neg Hardcovers(I_1, T, A), Price(I_2, C, P),$

\[ I_1 = I_2, C = 'gold\_member' \]

$V(I_1, T, A, I_2, DP) : \neg Book(I_1, T, F, A, SR), Book\_Price(I_2, S, RP, DP),$

\[ F = 'hardcover', SR = RP, S = 'B\&N' \]

Assertions. Since assertions are boolean queries, the satisfaction of an
intra-source assertion \( A \) by a source database \( DB \) (denoted \( DB \models A \)) means that \( A \) evaluates to true over \( DB \). Satisfaction of an inter-source assertion \( A \) by source databases \( DB_1, DB_2 \) (denoted \( DB_1, DB_2 \models A \)) is defined in the expected way.

**Queries and their Certain Answers.** Applications retrieve integrated data by issuing queries against the target schema. In this chapter, we restrict attention to queries expressed as unions of conjunctive queries with equalities and parameters. A parameterized query \( Q \) models the set of all non-parameterized queries in which \( Q \)'s parameters are replaced by arbitrary constants. As is typical in GLAV-based integration systems, we adopt as our query answering semantics the definition of certain answers to a query following the numerous works surveyed in [Len02, Hal00].

Starting from a set of source instances \( \overline{DB} = DB_1, \ldots, DB_n \) satisfying the source constraints, the set of corresponding GLAV registrations \( \overline{R} = R_1, \ldots, R_n \) does not define a single target instance. Instead there is in general a set \( \text{Targets}_{\overline{R}}(\overline{DB}) \) of possible target instances that satisfy the registrations and the target constraints \( \Delta_G \):

\[
\text{Targets}_{\overline{R}}(\overline{DB}) = \{ G | \bigwedge_{i=1}^{n}(DB_i, G) \models R_i \land G \models \Delta_G \}.
\]

The certain answers to a query \( Q \) (from now on referred to as simply “answers” to \( Q \)) are the common answers that we would get if we executed \( Q \) against each possible target:

\[
\text{Cert}_{\overline{R}}(\overline{DB}) = \bigcap_{G \in \text{Targets}_{\overline{R}}(\overline{DB})} Q(G).
\]

When there are no possible targets, we consider the set of certain answers as being empty.

Assume that B&N is the only source registered in the system as shown in Figure 2.4 and its database \( DB_1 \) has general and member price information for an Ullman book stored in the following two tuples: \texttt{Hardcovers} (“5”, “DB Systems”, “Ullman”) and \texttt{Price} (“5”, “gold members”, “$80”). Any target instance that satisfies the source’s registration will contain at least two tuples of the form:

\[
\texttt{Book} (“5”, “DB Systems”, “hardcover”, “Ullman”, X)
\]

\[
\]
Since the registration only specifies that the regular price of the book equals its suggested retail price without providing the price, the value of X will differ among the possible targets but within any single target it will have the same value in both tuples. Therefore X does not behave simply as a null. For instance, a query retrieving all books sold by B&N at the suggested retail price returns Ullman’s book regardless of the target instance (i.e. regardless of the specific value for X). Ullman’s book is therefore among the certain answers.

2.6.2 Inconsistency

Since sources register independently, their combined data could violate the target constraints. To help source owners avoid such cases, RIDE issues warnings on two levels of inconsistency, depending on whether it will always occur regardless of the data in the source database (definite inconsistency) or it will only appear if suitable data are present in the sources (potential inconsistency).

Potential Inconsistency. The integration system is in a potentially inconsistent state if for at least one instance of the source databases that satisfy the source integrity constraints and owner-accepted assertions, no instance over the target schema satisfies both the mapping and target constraints.

Consider the integration system consisting of n sources with schemas $S_1$, ..., $S_n$, databases $DB_1, ..., DB_n$ and corresponding registrations $R_1, ..., R_n$. Let the set of all accepted intra- and inter-source assertions be $A$ and denote with $DB \models A$ the fact that they are satisfied by the collection of source databases.

Formally, the integration system is potentially inconsistent iff

$$\exists DB_1, ..., DB_n \text{ over } S_1, ..., S_n \text{ s.t. }$$

$$DB_i \models \Delta_{S_i} \text{ for all } 1 \leq i \leq n, DB \models A, \text{ and } Targets_{R_1, ..., R_n}(DB_1, ..., DB_n) = \emptyset$$

Definite Inconsistency. The integration system is in a definitely inconsistent state if for any data in the registered sources satisfying the integrity constraints and assertions, there does not exist an instance over the target schema that satisfies both the mapping and target constraints.
Formally, the integration system is definitely inconsistent iff

\[ \forall DB_1, \ldots, DB_n \text{ over } S_1, \ldots, S_n \]

if \( DB_i \models \Delta_{S_i} \) for all \( 1 \leq i \leq n \) and \( DB \models A \), then

\[ \text{Targets}_{R_1, \ldots, R_n}(DB_1, \ldots, DB_n) = \emptyset \]

Examples of both inconsistency kinds were given in Section 2.5.2.

### 2.6.3 Levels of Self-Reliance

We formally define the levels of self-reliance of a registration w.r.t. an application query.

Assume that \( \bar{R} = R_1, \ldots, R_n \) is the set of registrations of the existing sources in the system and \( R_{n+1} \) is a registration of a new \( n+1 \)-st source. Let the intra-source assertions of source \( n+1 \) be denoted by \( A_{n+1}^{\text{intra}} \) and the inter-source assertions involving source \( n+1 \) be \( A_{n+1}^{\text{inter}} \). As above, the collection of all assertions pertaining to sources 1 through \( n \) is denoted \( A \).

**Self Sufficient.** The source registration \( R_{n+1} \) is *Self Sufficient* w.r.t. an application query \( Q \) if the \( n+1 \)-st source provides answers to \( Q \) even if the other registered sources leave the system.

For instance, if B&N’s owner extended the registration in Snapshot 1.4 of Figure 2.5 by providing an actual value for the regular price through a projection arrow from \( \text{Price.price} \), then B&N’s registration would be self-sufficient w.r.t. the query of Figure 2.3, since it would provide all attributes required by the query, thus contributing on its own at least one tuple to the query’s certain answer.

Formally, \( R_{n+1} \) is *Self Sufficient* w.r.t. \( Q \) iff

\[ \forall DB_{n+1} \text{ over } S_{n+1} \text{ s.t. } DB_{n+1} \models \Delta_{S_{n+1}} \text{ and } DB_{n+1} \models A_{n+1}^{\text{intra}} \text{ and } \]

\[ \text{Targets}_{R_{n+1}}(DB_{n+1}) \neq \emptyset; \]

\[ \text{Cert}_{Q,R_{n+1}}(DB_{n+1}) \neq \emptyset. \]

**Complementary.** Consider now a registration that is not Self Sufficient because the query answer will be empty if the other sources leave the system. If
however data from the corresponding source \( n + 1 \) can be combined with data from other sources in the system to create new answers to the query (which are not already contributed by the other sources without \( n + 1 \)’s help), we refer to this registration as \emph{Complementary}. Complementarity is usually enabled by primary key constraints on the target schema.

\emph{For example, PH’s registration in Snapshot 2.3 of Figure 2.6 is Complementary w.r.t. the query of our running example and B\&N’s registration of Snapshot 1.4 in Figure 2.5. Indeed PH’s registration contributes to the answer of the query only in the presence of the B\&N registration. Note how partial book information provided by both sources is combined to provide a query result. Since both sources provide information about the same book and Book.ISBN is a primary key (PK), the author provided by B\&N is merged with the suggested retail price exported by PH to give a single Book tuple. Furthermore, since B\&N sells the book normally at the suggested retail price, this merging also defines a value for the regular price at which B\&N sells the book. In this way PH’s registration creates a query answer that would not be there in its absence (i.e. it contributes to the answer of the query) but it relies on the presence of B\&N to do so (i.e. it is not Self Sufficient). Hence it is Complementary w.r.t. the query and B\&N’s registration.}

Formally, \( R_{n+1} \) is \emph{Complementary} w.r.t. \( Q \) and \( \bar{R} \) iff it is not Self Sufficient w.r.t. \( \bar{R} \) and

\[
\forall DB_1, ..., DB_{n+1} \text{ over } S_1, ..., S_{n+1}
\text{s.t. } DB_i \models \Delta_S \text{ and } \bar{DB} \models \mathcal{A} \cup A_{inter} \cup A_{intra} \text{ and}
\text{Targets}_{R_1, ..., R_n, R_{n+1}}(DB_1, ..., DB_n, DB_{n+1}) \neq \emptyset:

\text{Cert}_{Q_{R_1, ..., R_n, R_{n+1}}}(DB_1, ..., DB_n, DB_{n+1}) \supseteq \text{Cert}_{Q_{R_1, ..., R_n}}(DB_1, ..., DB_n).
\]

\emph{Unusable.} Finally, a source registration that is neither self-sufficient nor complementary is called \emph{Unusable}. 
2.7 Algorithms

RIDE’s backend is invoked after each user action. It takes as input the integration system’s parameters (registrations, source/target constraints and assertions) and the application query and carries out the following tasks: a) it checks for definite and potential inconsistency, b) it computes the current self-reliance level of the source registration and c) it makes suggestions on how to achieve the desired degree of self-reliance. In this section we describe the algorithms used to solve each of these problems.

The challenge lies in the data-independent nature of the checked properties, which calls for a way to reason about the properties of all instances that satisfy a set of constraints and assertions. It is of course infeasible to check Self Sufficiency by enumerating the infinitely many source databases and the infinitely many possible target databases. RIDE addresses this issue by building a single, canonically constructed source instance $\text{CanSource}$ and a corresponding possible target instance $\text{CanTarget}$. The instance $\text{CanSource}$ is over schema $\bigcup_i S_i$, and consists of the disjoint union of the canonical instances for each source. As proven by the theorems below, it suffices to check the consistency and self-reliance status on the canonical instances to ensure that they hold in general.

We start by presenting the construction of the canonical source and target instances, showing the decision procedures for inconsistency and self-reliance level next. We emphasize that all of these procedures are heavily based on evaluating queries on small (toy-sized) databases computed from the available constraints and assertions, which is what ensures their good response times in practice.

Canonical source and target instances. RIDE builds the canonical instances using the classical Chase procedure [AHV95]. While we do not describe the well-known chase in detail here, we show a lesser known algorithm for its implementation [FKMP05, DT03a]: it is based entirely on evaluating queries and therefore optimizable using the classical techniques employed in relational query optimizers, such as pushing selections into joins, join reordering, etc. [DT03a].

\begin{algorithm}
\textbf{Chase (by query evaluation)}
\begin{itemize}
\item \textbf{input:} instance $I$, set of constraints $\Delta$
\end{itemize}
\end{algorithm}
output: instance $J$ obtained by modifying $I$ to satisfy $\Delta$

begin
1. $J := I$
2. repeat
3. for each constraint $(U \subseteq V) \in \Delta$ and each tuple $t \in U(J) - V(J)$
4. modify $J$ (by adding the atoms in $V$'s body) to ensure $t \in V(J)$
5. until no new facts are added to $J$
6. return $J$
end

We show next how the canonical instances are computed using the chase. Notice that the chase is defined to work exclusively with constraints of the form $U \subseteq V$. While registration mappings exhibit this general form, the key reason enabling the applicability of the chase to our setting, which contains integrity constraints is the well-known fact that embedded dependencies (and therefore all common integrity constraints they express) can be expressed in the same way [FKMP05, DT03a].

**In our running example** Book.ISBN, shown underlined, is the primary key (PK) of target relation Book. This target PK constraint can be expressed as $(U_{PK} \subseteq V_{PK})$, where:

$$U_{PK}(I,T_1,F_1,A_1,SR_1,T_2,F_2,A_2,SR_2) : -
\quad \text{Book}(I,T_1,F_1,A_1,SR_1), \text{Book}(I,T_2,F_2,A_2,SR_2)$$


It is easy to see that a target database instance $G$ satisfies this constraint iff each pair of Book tuples that agree on the ISBN are identical (which is the usual definition of ISBN being the PK).

algorithm mkCanInst
Input: Set of source names $\mathcal{N}$
Output: A pair of canonical source and target instances

begin
1. $I :=$ the empty instance over combined source and target schemas $\bigcup_{i \in \mathcal{N}} S_i \cup G$
2. for each source query $Q$ appearing in
   a mapping constraint or assertion for some source $i \in \mathcal{N}$
   add a fresh copy of $Q$'s body (one tuple per query atom) to $I$
3. $J := \text{Chase}(I, \bigcup_{i \in \mathcal{N}} \Delta_{S_i} \cup \bigcup_{i \in \mathcal{N}} R_i \cup \Delta_{G})$
4. $\text{CanSource} :=$ restriction of $J$ to source relations
   $\text{CanTarget} :=$ restriction of $J$ to target relations
6. return (CanSource, CanTarget)
end

Notice that in algorithm mkCanInst, I is an instance consisting of a source and a target database pair. Line 2 adds to the source component of I data reflecting the non-emptiness of source queries in mapping constraints and assertions, since we restrict our attention to source instances satisfying such non-emptiness constraints. In Line 4, the chase with all available constraints has the following effect. The chase steps with the source integrity constraints $\bigcup_{i \in \mathcal{N}} \Delta_S_i$ infer all additional facts needed to make the source component of I satisfy the constraints. Chase steps with the mapping constraints $\bigcup_{i \in \mathcal{N}} R_i$ compute from the source part of I all tuples that must be exported into the target part of I in order to satisfy the registration mapping constraints. These tuples are then further chased with the target constraints $\Delta_G$, to obtain a compliant target instance. The source and target parts of the final chase result are returned as CanSource, respectively CanTarget.

Example: Consider an integration system consisting of the registrations of B&N and Prentice Hall shown in Snapshots 1.4 and 2.4 of Figure 2.5 and 2.6 re-
spectively, together with the accepted assertions shown in the corresponding previous Snapshots (for the B&N registration RIDE uses only the assertion of Snapshot 1.3 and not the one of Snapshot 1.2, since the first extends the latter). In this case we get the canonical source instance depicted in Figure 2.7. The description next to each tuple indicates how the tuple was generated. For example the first tuple in table Hardcovers was introduced due to the non-emptiness of the left hand side of B&N’s mapping. If we assume for now that we have no constraints on the target schema, the corresponding canonical target instance is shown at the bottom of the same figure. Color-coding and text explain how each tuple was created. For example the first tuple in Book (colored white) was created from the identically colored first tuples of Hardcovers and Price through the B&N mapping. Values ending with a “*” are new values created in the target, because they were provided neither through a projection arrow nor a target selection condition in the corresponding mapping.

2.7.1 Deciding Inconsistency

At each interaction step, RIDE’s backend checks for both flavors of inconsistency. Definite inconsistency is the more vital one to be detected since it will unavoidably lead to inconsistency regardless of the contents of the databases. Algorithm isDefInconsistent presented below is guaranteed to detect this inconsistency. Interestingly, potential inconsistency turns out to be undecidable (Theorem 2 below), so RIDE uses a heuristic test, emitting a warning upon detection, and possibly failing to detect it. Fortunately, potential inconsistency is the more benign flavor, in the sense that it is more likely to be a conservative theoretical problem which does not necessarily have to occur in practice. Indeed, it arises whenever two owners provide entities into the same table that has a primary key: one can always populate the source databases to obtain agreement on the PK attributes and disagreement on the others, but this is not unavoidable. This observation is precisely what RIDE uses to warn source owners of potential inconsistency.

**Definite Inconsistency.** An integration system is definitely inconsistent
if for all source instances that satisfy the source constraints and assertions the set of possible targets is empty. This means that for any source instance, the target instance cannot be created. Looking back at the chase, this can only happen when the creation of CanTarget causes conflicts. One such conflict can appear because of target constraints (e.g. PKs) that cause an equality between two different constants. Interestingly, we can formally prove (see Theorem 1 below) that this is the only case that can prevent the existence of a target instance, and therefore RIDE detects Definite Inconsistency by employing the following procedure:

**algorithm IsDefInconsistent**

**Input:** Sources 1, ..., n

**Output:** true iff there is definite inconsistency

**begin**

(CanSource, CanTarget) := mkCanInst({1, ..., n})

if CanTarget contains an equality between distinct constants

then return true, else return false

*end*

**Example:** If Book.ISBN is a PK, then, since the third and fourth Book tuples in Figure 2.7 have the same ISBN, the chase will introduce an equality between ‘hardcover’ and ‘paperback’. This indicates a definite inconsistency, since a book has to be both paperback and hardcover. Recall that this is the inconsistency that RIDE explained to the user in Snapshot 2.4 of Figure 2.6.

The fact that algorithm IsDefInconsistent is a sound and complete decision procedure for definite inconsistency follows from:

**Theorem 1.** The registrations $R_1, \ldots, R_n$ lead to definite inconsistency if and only if $\text{mkCanInst} \{1, 2, \ldots, n\}$ creates an equality between distinct constants.

**Potential Inconsistency.** It turns out that there is no algorithm for deciding potential inconsistency:

**Theorem 2.** Potential Inconsistency is undecidable.

The proof (contained in Appendix A) is by reduction from the Post Correspondence Problem.
Therefore RIDE employs the following best-effort procedure to zoom in on the most obvious inconsistency causes: whenever two mappings provide tuples into the same target table $R$, both providing all attributes of the primary key of $R$, a potential inconsistency is signaled to the user, who can decide whether she expects her source to export data with the same key as the partner source, but with disagreement on the non-key attributes.

2.7.2 Deciding Self-Reliance Levels

**Self Sufficient.** We decide Self Sufficiency using the following procedure:

```plaintext
algorithm IsSelfSufficient
Input: registration $R_{n+1}$; application query $Q$
Output: true iff $R_{n+1}$ is Self Sufficient w.r.t. $Q$
begin
    $(CanSource, CanTarget) := mkCanInst([n+1])$
    if $Q(CanTarget) \neq \emptyset$ then return true, else return false
end
```

The correctness of this algorithm is given by Theorem 3 below, which follows from the definition of Self Sufficiency and from a classical result which states that the certain answers to $Q$ can be computed by running $Q$ over $CanTarget$ [AD98, Len02, FKMP05].

**Theorem 3.** A registration $R_{n+1}$ is Self Sufficient w.r.t. a query $Q$ iff $Q(CanTarget) \neq \emptyset$.

**Complementary.** Recall that a registration $R_{n+1}$ is Complementary w.r.t. a query $Q$ and existing source registrations $\bar{R}$ iff there is a certain answer tuple in the presence of both $R_{n+1}$ and $\bar{R}$ that would be missed in the absence of $R_{n+1}$. Since the certain answers of a query can be computed by running it over the corresponding canonical instance, it suffices to check whether $Q$’s answer on the $CanTarget$ constructed through all registrations (including $R_{n+1}$) strictly includes $Q$’s answer on the $CanTarget$ built from the existing source registrations only. The resulting algorithm and the theorem that guarantees its correctness are shown below:
algorithm IsComplementary
Input: existing registrations $R_1, \ldots, R_n$; new registration $R_{n+1}$; query $Q$
Output: true if $R_{n+1}$ is Complementary w.r.t. $R_1, \ldots, R_n$ and $Q$
begin
  $(\text{CanSource}, \text{CanTarget}) := \text{mkCanInst}([1, \ldots, n])$
  $(\text{CanSource}', \text{CanTarget}') := \text{mkCanInst}([1, \ldots, n, n+1])$
  if $Q(\text{CanTarget}') \supseteq Q(\text{CanTarget})$ then return true, else return false
end

The correctness of algorithm IsComplementary follows from:

Theorem 4. A registration $R_{n+1}$ is Complementary w.r.t. a query $Q$ and existing registrations $R_1, \ldots, R_n$ iff the result of $Q$ on the canonical target instance corresponding to $R_1, \ldots, R_n$ is strictly contained in the result of $Q$ on the canonical target instance for $R_1, \ldots, R_{n+1}$.

2.7.3 Computing Suggestions

As described in Section 2.5, RIDE’s suggestion component operates in two steps. First, it computes the different sets of attributes that the current mapping can provide to reach the desired self-reliance level and shows them on the right pane of the interface. Then, after the user selects a set and clicks on one of its attributes, RIDE computes and displays actions that can lead to its provision.

Computing sets of missing attributes. A source can contribute to an application query in many different ways (which involve providing different sets of attributes), depending on which data already in the system it decides to complement (see first example in Section 2.5.1). To compute the different sets of attributes that can be provided, RIDE starts from the observation that each certain answer tuple corresponds to a match of $Q$’s body against the canonical target instance. Therefore, in order for the currently constructed mapping to contribute at least one tuple to $Q$’s certain answer, it must generate new CanTarget tuples that, together with the tuples in CanTarget from the other mappings, serve as $Q$’s match. If such is the case, no new suggestions are needed. Otherwise, RIDE looks for partial matches of $Q$’s body against CanTarget, with the intention that for each partial match, the matched attributes of the query are contributed by
the registrations so far, and the unmatched ones will be provided by the mapping under construction.

**Computing suggestions for a single attribute.** When the owner clicks on a missing (i.e. unmatched) attribute, RIDE generates suggestions for it, searching through a list of potential actions shown below. An action is only suggested if it can be followed up with some sequence of actions that extend the registration to a consistent one, with the intended self-reliance. To find such a sequence, RIDE carries out the candidate action tentatively (extending the mapping accordingly) and then tries recursively to perform further potential actions to provide the remaining attributes. If the desired self-reliance is reached without encountering an inconsistency, then the candidate action is suggested, otherwise the search backtracks.

Essentially, during this search RIDE starts from the partial match of the query into *CanTarget* that generated the attribute set picked by the owner, and attempts to extend it to a total match. RIDE considers the following potential actions to this end:

1. **Projection arrows and target conditions:** RIDE checks if the selected attribute can be provided directly through a projection arrow from some source attribute or through a target condition.

2. **Source conditions and joins:** Since source conditions (resp. joins) limit the amount of information exported and hence do not usually lead to increase of self-reliance, they are only considered if the query contains them (as illustrated in Snapshot 1.2 and 1.3 respectively) and they do not map into the canonical target instance.

3a. **Intra-source assertions due to query selections:** If the query contains a constant in the clicked target attribute and this attribute is already mapped into from a source attribute, RIDE generates an assertion that for some source tuple the corresponding attribute value equals the query constant. This led to the assertion in Snapshot 1.2.

3b. **Intra-source assertions due to query joins:** Similarly to query selections, if the query involves a join between two target attributes provided by two source
attributes, RIDE generates an assertion that the source contains tuples in which
the source attributes have the same value. We saw such an assertion in Snapshot
1.3.

4. Indirect provision using data merging and inter-source assertions. RIDE
also makes suggestions for providing an attribute indirectly through another at-
ttribute. Such indirect attributes are detected when the partial query match against
CanTarget matches the intended attribute into a value that does not appear in any
registration or assertion, being instead freshly created during the construction of
the canonical target instance (see Retail2* in Figure 2.7). These values are known
as labeled nulls [FKMP05]. All occurrences of the same labeled null mark attribute
occurrences sharing the same (unspecified) value. RIDE attempts to provide con-
crete values for a labeled null by suggesting the provision of data that merge with
any of its occurrences. To achieve this merging, RIDE also suggests actions to
provide values into the keys determining these attributes. In our example, the
two occurrences of labeled null Retail2* led to the suggestions (in Snapshot 2.1) to
indirectly provide attribute Book.sug_retail instead of Book_price.reg_price, together
with the key Book.ISBN and to accept the assertion in Snapshot 2.2.

2.7.4 Complexity

Termination of the Chase. The property of weak acyclicity of a set of
constraints is sufficient to guarantee that any chase sequence terminates [FKMP05,
DT03b]. Very roughly, the restriction requires the FK constraints to not create
cyclic “refers-to” relationships between the attributes in the schema. In our GLAV
scenarios, weak acyclicity holds trivially in the cases (among many others) where
(i) the source and target schemas contain only PKs, or (ii) they contain both PKs
and FKs, but have a star, chain, or chain-of-stars (snowflake) shape [DT03a]. Of
course, for arbitrary constraints the chase may not terminate, as termination is
undecidable [AHV95].

Complexity of creating the target instance. Since all algorithms in-
volve creating the canonical target instance, they are affected by the complexity
of mkCanInst. We consider the typical case in which the integrity constraints
are (weakly acyclic) sets of primary and foreign keys, the target schema (and constraints) are fixed and only the source schemas and their registrations vary.

We introduce the following notation: \( N_S \) is the number of relations in the combined source schemas; \( e \) is the maximum length (in number of relational atoms) of a source query appearing in any registration mapping constraint; \( b_R \) is the number of mapping constraints in which source relation \( R \) is mentioned; \( b \) is the maximum \( b_R \) over all source relation names \( R \); \( t \) is the maximum number of relational atoms per target query in a mapping constraint.

Finally, given a primary key PK on target relation \( R \), let \( k_{PK} \) be the maximum number of distinct tuples, all agreeing on some value \( v \) for the PK attribute, which could be chased into \( R \) during the canonical instance construction. Then we denote with \( k \) the maximum \( k_{PK} \) over all target primary keys. Let \( N_v \) denote the number of distinct key values \( v \) as above.

An analysis of the chase run-time behavior yields that \texttt{mkCanInst} runs in worst-case time \( O(N_S b^e t + N_v k^2) \). Note that the exponent \( e \) is independent of the number of sources. It pertains to the largest number of source tables involved in a single mapping, a typically small value bounded by the size of each individual source schema, and more effectively, by the owner’s limited capacity of comprehending complex mapping constraints. Indeed, whenever possible, owners prefer to split the registration into several small mappings rather than wielding a single large one.

The \( k^2 \) term is due to chasing with the key constraint for PK on \( R \), which requires self-joining \( R \) on the PK [AHV95]. While \( N_v \times k \) is worst-case bounded by a polynomial in the combined size (number of variables) of all source queries appearing in registration mappings, in practice this is a small entity, as it really reflects the cases in which users put the same constant selections on key attributes or (via source selections) on attributes which finally end up providing values of key attributes. These registrations are unlikely: they would correspond for instance to the user restricting her registration to provide only data about the book of ISBN '123'. The only other factor contributing to the size of \( k \) and \( N_v \) are assertions, of which we expect the user to accept only a small number.
2.7.5 Experimental Evaluation

What we measured. To measure RIDE’s response time and see how it scales for large number of sources, we created a synthetic, yet typical integration scenario, consisting of several existing source registrations and a representative application query over the target schema. In this setting we ran a script simulating all possible interactions with RIDE by systematically following the tool’s suggestions until complementarity is reached. During this process, we measured the average and maximum time to generate the new suggestions for the subsequent interaction step.

The target schema. To create a realistic integration scenario, we used a target schema arranged as a snowflake (i.e. stars of stars). A star consists of a center table (with a PK) and a number of ray tables, pointing to the center via FKs. The snowflake is created by each ray being in turn the center of another star and so on. This design emerges naturally when normalizing wide universal relations as used in sciences and is also prevalent in data warehousing. It is also a more realistic setting obtained by mixing the two extremes of synthetic schema shapes used in typical benchmarks: chain- and star- shaped schemas. In the snowflake schema, the central table usually holds the required attributes of a concept (e.g. organism in sciences or business concept in data warehouses) and the rays hold optional sets of data characterizing this concept (e.g. sets of experiments to measure a given property of the organism). Recall that, although for simplicity we used only PK constraints in our running example, RIDE supports both PKs and FKs and more expressive constraints out of the class of embedded dependencies. The inclusion of FKs in our target schema stresses the tool by increasing the size of the canonical target instance generated by the chase. Indeed, a single tuple \( t_1 \) created in the target through a mapping constraint leads to the creation of a new tuple \( t_2 \) referenced by \( t_1 \) via the FK, which in turn yields a new tuple (if any) referenced by \( t_2 \), etc.

The source schemas. As source schemas we used single tables. For every star’s ray in the target schema we created a new source that maps into both the ray and the center of the star.
A family of configurations. Our setting is scaled by two parameters $r$ and $d$. If we represent the target schema as a directed graph where each node corresponds to a table and each edge from table $A$ to table $B$ corresponds to a FK in $A$ referencing $B$, then we define as the diameter $d$ of the snowflake the length of the longest directed path in the graph. Additionally $r$ denotes the number of rays of each star. A snowflake of diameter $d$ in which each star has $r$ rays contains $\frac{r^d + 1}{r-1}$ tables. The number of sources is $\frac{r^d + 1}{r-1} - 1$ and both their number as well as their overlap increases with $d$ and $r$. Figure 2.8 depicts the schema for $d = 2$ and $r = 3$ and a source registration providing the two shaded target relations.

The platform. The measurements were conducted on a PC with a Pentium 4 3.2 GHz, MS Windows XP Pro and 1GB RAM.

The results. For increasing values of the parameters, we explored the tree
of all possible interaction runs to contribute to a query performing a 3-way join over the snowflake. Although the query had 12 attributes, RIDE correctly asked only for the required attributes (which for our query were 5). In some cases this number was even smaller as the tool exploited merging and borrowed values from other sources. The number of required attributes also defines an upper bound on the number of interaction steps until complementarity is reached. Since any required attribute can be provided through two actions (adding an arrow or selection and potentially accepting an assertion), the depth of the interaction is at most twice the number of required attributes (10 in our setting).

Figure 2.9 shows RIDE’s average and maximum response time w.r.t. the number of sources in the system (generated by using $d = 2$ and $r$ ranging from 1 to 15). The highest values in the graph are for 240 sources ($d = 2$, $r = 15$) with RIDE taking in the worst interaction sequence a maximum of 1.223 sec to respond. Its average response time was even better: 0.615 sec. Out of this time the generation of candidate suggestions was negligible, with most of the time spent in checking whether such a suggestion leads to the desired self-reliance. These results show that RIDE’s response time meets the needs of interactive tools even for complex target schemas and sufficiently many sources to preclude global overview.

### 2.8 Related Work

RIDE adopts the GLAV formalism introduced in the context of open-world integration systems [FLM99, Hal00, Len02], later used in data exchange [FKMP05] and peer-to-peer integration systems [HIM+04, FKMT05]. However, none of these lines of work addresses autonomous source registration, levels of self-reliance, or visual guidance towards them.

Efficient Implementations of the chase algorithm based on query evaluation are reported and evaluated in [DT03a, FKMP05, CCGL02].

Recently, in Cimple/DBLife [DRC+06] it was suggested that a central authority integrates community data from the web through tools that semi-automatically retrieve and integrate unstructured data in a best-effort way (which may lead
to inconsistencies or wrong data). Our approach is orthogonal, being suitable for
communities willing to integrate structured data in a precise way. Since the source
registration has to be done manually, we help the central authority by delegating
this job to the individual community members.

Commercial data integration tools such as IBM WebSphere QualityStage [web]
detect primary key violations (so-called duplicate tuples) by inspecting the under-
lying data instances. Other projects allow inconsistencies but rewrite applica-
tion queries to take into account only the consistent part of the database [FM05,
ABC99], or to compute probabilities for each of the inconsistent duplicate tu-
ples [AFM06]. Our focus on inconsistency is complementary, emphasizing preven-
tion and explanation at registration time rather than detection and resolution at
run time.

2.9 Extensions and Future Work

In this section we present two possible extensions of our work on au-
tonomous source registration; support for evolution of communities and support
for contribution to more than one queries.

2.9.1 Supporting Evolution of Communities

In our framework a community is started by an initiator who designs its
target schema. Sometimes the initiator is a consortium agreeing on a common
schema. More commonly we envision the emergence of ad hoc communities whose
initiator (possibly an individual) decides the schema without seeking source owner
approval. Source owners join the community as it gains popularity (just as online
communities like blogs grow). Such communities may evolve. Evolution may
include both changing the target schema of a single community (to make it adapt
to new needs) and coalescing of several ad hoc communities that have emerged on
the same topic into a larger community.

The community-based integration architecture can support both types of
evolution through techniques studied extensively in [VMP04, YP05, DPT06] as
explained next. We will start by presenting the case of schema evolution within a single community and then we will show that coalescing of communities can be reduced to the former.

When the initiator evolves a community’s target schema, this affects both the legacy source registrations and application queries. RIDE can help keep the maintenance task lightweight by relieving the initiator from the need to know anything about sources and their mappings. To this end we propose existing techniques to automate the translation of mappings and queries to the new schema. In particular, mappings are adapted to the new target schema using techniques presented in [VMP04, YP05]. Similarly, the queries are rewritten against the new target schema by modeling the schema evolution as a mapping between the old and the new target schema and using solutions on rewriting queries under constraints (see [DPT06]). A source owner can subsequently call RIDE as usual to adjust the contribution of the new mappings to the new application queries. RIDE thus assists in delegating the non-scalable part of schema evolution to the individual source owners.

The coalescing is supported by the same techniques used for schema evolution within a single community. When initiators merge their communities into a larger one, they design the new community’s target schema (which might be either a new schema or one of the schemas of an existing community) and they map the individual community schemas to it. Subsequently the same techniques as before can be used to adapt the mappings and queries to the new target schema.

### 2.9.2 Contributing to a List of Queries

In this chapter, we focused on guiding the registration to contribute to a single query. However, our approach lends itself to generalization to a list of queries: the owner visits each query in turn, adding mappings until the desired level is reached for the current query. It is easy to see that adding a mapping cannot lower (but could increase) the already achieved self-reliance level of previously visited queries. RIDE is guaranteed to avoid generating suggestions for providing attributes needed by a query if they have already been provided for a previous
query. This is a consequence of our solution being based on matching the query against the canonical target instance, which essentially grows with each added mapping, thus increasing the matching opportunities for the new query. The local minimization thus achieved for the publishing cost depends however on the historic order in which queries were visited by the owner. A global consideration of the entire query list could potentially yield more minimization opportunities, and we intend to address this question in future work.

2.10 Conclusions

We target communities of data owners motivated to publish their data autonomously into the community schema. Our aim is to enable owners to autonomously negotiate the trade-off of self-reliance in making their data visible to applications, versus minimization of the publishing cost. To this end we define 3 degrees of self-reliance for contribution, and introduce RIDE, a visual tool that guides the owner by suggesting which attributes to provide. RIDE guarantees that, by following its suggestions, the user will arrive at a registration of the desired self-reliance level, incurring the cost for providing only essential attributes, and avoiding inconsistency. Our evaluation shows that the algorithms for checking consistency and self-reliance and for generating suggestions, scale well with the numbers of sources. A demo is available at http://db.ucsd.edu/ride.

This chapter is, in part, a reprint of the research paper published in VLDB 2008 by Yannis Katsis, Alin Deutsch and Yannis Papakonstantinou and contains material published in a demonstration paper in VLDB 2008 by Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou and Keliang Zhao.
Chapter 3

Self-Reliance Levels Revisited

In Chapter 2, we presented RIDE; a tool for autonomous source registration and explained how it is structured around the notion of the self-reliance levels. In this Chapter we revisit the self-reliance levels and provide alternative definitions of them. As we will explain next, these definitions (which came chronologically earlier than the ones presented in the context of RIDE) are not suitable for a source registration tool. However they constitute an important theoretical contribution, providing a comprehensive survey of the self-reliance problem. This chapter is structured as follows: Section 3.1 provides some background information and Section 3.2 formalizes the setting. Sections 3.3 and 3.4 introduce the new definitions of the self-reliance levels formally and through examples, respectively. Section 3.5 contains the main results of the chapter: It shows decidability of the new self-reliance levels are decidable under different classes of target constraints. Finally, Section 3.6 discusses related work and Section 3.7 concludes the chapter.

3.1 Background

In Chapter 2 we showed how we can capture different trade-offs between reliance on other sources to contribute to a query and the associated cleaning effort through different self-reliance levels. The self-reliance levels were carefully crafted to ensure that they are suitable for a source registration tool. In particular, if you recall the definitions of the self-reliance levels presented in Section 2.6.3, they use
a *universal* quantification over the source databases that satisfy the assertions and source constraints. This is crucial to avoid re-checking consistency and self-reliance levels upon every update to source tables, as well as the annoying notifications to source owners. Instead, all there is to check is the preservation of the integrity constraints and assertions, which can be done through classical solutions [GSUW94].

In this Section we present an alternative set of definitions of self-reliance levels that we have created before designing RIDE. Instead of using universal quantification over the source databases, they use *existential* quantification. This corresponds to *potential* self sufficiency and complementarity, as opposed to the *definite* flavor checked by RIDE. Naturally these new definitions call for a completely new set of algorithms, which we present in this chapter.

For example, consider a car shopping portal. Multiple car dealers contribute advertisements, while at the same time third parties provide reviews, such as the “Blue Book” (http://www.kbb.com). Then one can easily build the, say, “Great Auto Deal” web application that looks for cars of a user-provided make and type, selling for less than 10% of their Blue Book value.

In contrast to the three degrees of self-reliance defined for RIDE, we distinguish four qualitative categories of self-reliance of a source schema and mapping pair with respect to a given application query:

- **Self Sufficient**: Given the mapping, the source can contribute certain answers to the query, even in the absence of any other source. For example, consider a query that asks for cheap cars, regardless of how their value compares to the Blue Book. Then a car dealer’s contribution is self-sufficient, assuming it provides the price and the other attributes of interest to the query.

- **Now Complementary**: The source can contribute certain answers but only because other sources provide complementary data. For example, consider a query that asks for cars that are 10% cheaper than their Blue Book value and assume that the Blue Book is already registered. Then the car dealer’s source is now complementary since it relies on an existing source in order to contribute.
• *Later Complementary:* The source cannot contribute certain answers currently but it is possible to contribute if an appropriate new source is registered in the future. For example, consider the case of “now complementary” above whereas a source such as the Blue Book, providing typical prices, has not yet been registered. Then the dealer’s source is later complementary.

• *Unusable:* The source inherently does not contribute.

As we will see later, the complementary level defined for RIDE corresponds essentially to the definite version of the now complementary definition above. The later complementary definition was left out from RIDE, since it would be too weak and therefore would never hold: In particular, it would require that first, the source is not now complementary and second, for all instances of the current source satisfying the assertions and source constraints and all instances of some future source (without any assertions), the current source contributes to the application query by relying on the future source. This obviously never holds, since the possible instances of the future source would also include the empty instance (since no assertions are imposed on it).

**Contributions.** We study the problem of deciding the self-reliance level when the mappings are given by source-to-target constraints in the language of embedded dependencies, and the application queries are unions of conjunctive queries with equalities. Figure 3.1 summarizes our decidability results. Each row corresponds to one of the self-reliance levels, while columns correspond to various assumptions on the set $\Delta_G$ of constraints on the global schema $G$. PKs stands for primary keys and RICs for referential integrity constraints, all of which are expressible in the more general class of embedded dependencies [AHV95]. Our undecidability results therefore carry over to this class. Question marks denote open problems, which we conjecture undecidable.

Our decidability results for self-sufficiency and now-complementarity are due to a reduction to checking satisfiability and containment of a certain rewriting $R$ of an application query $Q$. $R$ is expressed against the local schemas and it returns the certain answers of $Q$. Developing a relevant containment test is not straightforward. Previous work [DGL00] describes only algorithms which yield a
<table>
<thead>
<tr>
<th>$\Delta_g$</th>
<th>none</th>
<th>PKs</th>
<th>PKs + RICs</th>
</tr>
</thead>
<tbody>
<tr>
<td>suff.</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>now comp.</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>later comp.</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
</tr>
<tr>
<td>unusable</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure 3.1: Decidability of categorizing registrations (see Section 3.5 for complexity results)

certain rewriting as a recursive Datalog program when functional dependencies (FDs) are given on the global schema. These algorithms do not serve our purpose, as containment of recursive Datalog programs is undecidable [AHV95]. However, we show (in Theorem 12) that, if each relation in the target schema allows at most one FD,\(^1\) then there exists an equivalent, nonrecursive rewriting expressed as a union of conjunctive queries with equalities (UCQ\(^e\)). This case is quite common in practice, covering that of primary keys and BCNF schemas. Moreover, we show how the nonrecursive certain rewriting under the Open World Assumption can be constructed by reusing an algorithm for finding exact rewritings under the Closed World Assumption. Our result is therefore of independent interest as an extension on prior work in finding certain rewritings, and also maximally-contained rewritings [DGL00] shown in [AD98] to be equivalent in our setting to certain rewritings. The result sheds additional light on the connection between exact rewritings in the closed world assumption and certain rewritings in the open world assumption.

### 3.2 Preliminaries

In this chapter we employ a similar setting to the one presented in Chapter 2 without however taking into account source constraints or assertions. Please note that the definitions and notation used vary slightly between the two chapters.

\(^1\)This includes the case of several FDs which can be summarized into a single one using Armstrong’s axioms [AHV95].
Constraint-based Data Integration. We consider systems which integrate a collection of $n$ data sources, where for each $1 \leq i \leq n$, source $i$ has a schema $S_i$ and extent $DB_i$. The local sources contribute to a global, integrated database $G$ of global schema $\mathcal{G}$, satisfying the set $\Delta_\mathcal{G}$ of integrity constraints expressed in terms of $\mathcal{G}$ (denoted $G \models \Delta_\mathcal{G}$). The contribution of source $i$ to the global database is described by a set $M_i$ of mapping constraints over the combined schemas $S_i$ and $\mathcal{G}$. Specifically, the constraints are of the form $U \subseteq V$, with $U$ and $V$ queries against schema $S_i$, respectively $\mathcal{G}$. Intuitively, these constraints specify that, given a local database $DB_i$ and a global database $G$, the local data identified as $U(DB_i)$ is visible among the global data identified by $V(G)$: $U(DB_i) \subseteq V(G)$. Notice that there are no containment statements in the opposite direction because an individual local source owner cannot know what other sources contribute to the global database and therefore cannot presume to contribute all global data. For instance, a local Toyota dealership’s data source may contribute its cars to a state-wide car database, but cannot claim to offer all globally accessible car offers (including other brands). This is consistent with the open world assumption [Hal00, Len02].

The global database is described indirectly as a database $G$ which satisfies the integrity constraints in $\Delta_\mathcal{G}$. Moreover, together with each local source $DB_i$, $G$ satisfies the mapping constraints in $M_i$. This is denoted $DB_i, G \models M_i$ and defined as:

$$DB_i, G \models M_i \iff \bigwedge_{(U \subseteq V) \in M_i} U(DB_i) \subseteq V(G).$$

Given $DB_i$, $M_i$ does not fully identify $G$, as it only states that $G$ must hold part of $DB_i$’s data, leaving unspecified what else $G$ may contain. There are therefore (potentially infinitely) many possible global databases which, together with $DB_i$, satisfy $M_i$. We think of $M_i$ as inducing a mapping from each data source $DB_i$ to its set of possible global databases. We denote the set of targets of $DB_i$ through this mapping as:

$$Targets_{M_i}(DB_i) = \{G : (G \models \Delta_\mathcal{G}) \land (DB_i, G \models M_i)\}.$$
The set of possible global databases defined by a collection of sources $\overline{DB} = DB_1, \ldots, DB_n$ and their sets of mapping constraints $\overline{M} = M_1, \ldots, M_n$ consists of the global databases which are simultaneously targets of each $DB_i$ under $M_i$:

$$\text{Targets}_{\overline{M}}(\overline{DB}) = \bigcap_{i=1}^{n} \text{Targets}_{M_i}(DB_i).$$

Since the set of possible global databases can potentially be infinite, applications can only inspect them indirectly, by posing queries against them. Given an application query $Q$ against the global schema $G$, the system returns only the certain answers to $Q$. These are defined as tuples appearing in the result of $Q$ on each possible global database. We denote the set of certain answers to $Q$ with $\text{cert}_{\overline{DB}}(Q)$ and define it as:

$$\text{cert}_{\overline{DB}}(Q) = \begin{cases} \emptyset, & \text{if } \text{Targets}_{\overline{M}}(\overline{DB}) = \emptyset \\ \bigcap_{G \in \text{Targets}_{\overline{M}}(\overline{DB})} Q(G), & \text{otherwise} \end{cases}$$

For an existing open-world system in which source contributions are described using mapping constraints (called source-to-target constraints there) and in which application queries are answered under the “certain answers” semantics, see IBM’s Clio system [MHH+01].

**Queries.** A term is a variable or constant. By $\bar{x}$ we denote a finite sequence of terms $x_1, \ldots, x_k$. The language of conjunctive queries with equalities ($\text{CQ}^=$) consists of expressions of the form $Q(\bar{x}) : \ell_1(\bar{x}_1), \ldots, \ell_n(\bar{x}_n)$ where we define the head and body of $Q$ to be the parts to the left and to the right of the $: -$ respectively. Each $\ell_i(\bar{x}_i)$ in $Q$’s body is a literal, i.e., an atom $R(\bar{x}_i)$ or an equality $x_g = x_h$ with $\bar{x}_i = x_g, x_h$. The language $\text{UCQ}^=$ denotes all unions of $\text{CQ}^=$ queries.

**Constraints.** For a given query language $\mathcal{QL}$, we consider the corresponding constraint language

$$\text{IC}(\mathcal{QL}) := \{(U \subseteq V) : U, V \in \mathcal{QL}\}.$$  

Such constraints express the containment of query $U$ in query $V$ and are equivalent in expressive power to embedded dependencies [AHV95] when $\mathcal{QL} = \text{CQ}^=$.
Embedded dependencies can express standard integrity constraints such as primary keys (PKs) and referential integrity constraints (RICs). For example, the following IC(CQ^=) constraint on the global schema of Figure 3.2 states that Model and Seller provide a combined PK for ads in the ad relation: \( (U \subseteq V) \), where

\[
V(M, S, P, P) :\geq \text{ad}(M, S, P) \quad \text{and} \\
U(M, S, P_1, P_2) :\geq \text{ad}(M, S, P_1), \text{ad}(M, S, P_2).
\]

RICs are particular cases of IC(CQ^=) constraints stating inclusions between projections of relations. For example, the following RIC states that all ads refer via their Model attribute to cars whose details are published in the car relation: \( (U \subseteq V) \), where

\[
U(M) :\geq \text{ad}(M, S, P) \quad \text{and} \\
V(M) :\geq \text{car}(M, C, D, B).
\]

In this chapter, we assume that all application queries belong to UCQ^= and that all constraints belong to IC(CQ^=). This is the case in all our examples as well.

### 3.3 Self-Reliance Levels

We introduce the notion of contribution of a source’s extent to a given application query, as well as four qualitative degrees of potential self-reliance of a source registration relative to the other sources registered in the system.

Each individual source owner registers his local source \( i \) to the integration system by declaring the source schema \( S_i \) as well as the set of mapping constraints \( M_i \). We call the pair \( R_i = (S_i, M_i) \) a source registration. Let \( \overline{R} = R_1, ..., R_n \) be the list of registrations of all sources present within the system and \( \overline{D} = D_1, ..., D_n \) the corresponding source instances. Also, let \( Q \) be a query formulated by an application against global schema \( G \).

**Contribution of a source to a query.** If \( R_{n+1} = (S_{n+1}, M_{n+1}) \) is the registration of a new source and \( D_{n+1} \) an instance of it, then the contribution of the new source to \( Q \) is denoted \( \text{constr}^Q_{\overline{R}, \overline{D}}(R_{n+1}, D_{n+1}) \) and corresponds to the certain answers to \( Q \) which are not provided by the already registered \( n \) sources:

\[
\text{constr}^Q_{\overline{R}, \overline{D}}(R_{n+1}, D_{n+1}) = \text{cert}^M_{\overline{D}, D_{n+1}}(Q) \setminus \text{cert}^M_{\overline{D}}(Q)
\]
Potential self-reliance of a registration. For the benefit of the source owner, we determine whether the source has any chance to contribute (now or in the future) any certain answers to some given application query $Q$. The definitions are data independent by existentially quantifying the source extents. Specifically, we say that source registration $R_{n+1}$ is

- **Self Sufficient** (written as $Suf_Q(R_{n+1})$)
  iff there is an extent of source $n + 1$ (not necessarily the current one) for which it can contribute certain answers to $Q$ even in the absence of any other source. Formally, iff there is a source extent $D_{n+1}$ such that $\text{contr}^Q_{\emptyset,\emptyset}(R_{n+1}, D_{n+1}) \neq \emptyset$.

- **Now Complementary** ($NComp^Q_R(R_{n+1})$)
  iff source $n + 1$ on its own brings an empty contribution regardless of its extent, but it could contribute in cooperation with the other sources, provided appropriate extents for them and for source $n + 1$:
  - not $Suf_Q(R_{n+1})$ and
  - there are source extents $\{D_i\}_{1 \leq i \leq n+1}$ such that $\text{contr}^Q_{R, D_1, \ldots, D_n}(R_{n+1}, D_{n+1}) \neq \emptyset$

- **Later Complementary** ($LComp^Q_R(R_{n+1})$)
  iff source $n + 1$ is currently not now complementary given the registrations $R$, but could become so given future source registrations $R'$:
  - not $NComp^Q_R(R_{n+1})$ and
  - there is a list of registrations $R'$, s.t. $NComp^Q_{R'}(R_{n+1})$

- **Unusable** ($Unusable^Q_R(R_{n+1})$)
  iff source $n + 1$ cannot contribute now (even if its own extent or the extents
of the existing sources were to change) or in the future (if any other sources register within the system):

- not $Suf_Q(R_{n+1})$ and
- not $NComp_Q^R(R_{n+1})$ and
- not $LComp_Q^R(R_{n+1})$
3.4 Examples

In this section we introduce our running example and use it to illustrate the levels of self-reliance in both the absence and presence of target constraints.

Consider the creation of a car portal that integrates information about new cars for sale. The global schema $G$, designed by the portal owner, is shown on the right in Figures 3.2a-i and 3.2b-i. The left side contains the schema $S_1$ of the already registered source 1 and respectively, $S_2$ of a new source 2, whose contribution we want to check. All schemas are shown in a tree-format. Bullets represent relations, hyphens correspond to attributes and dashed lines connect a relation with its attributes. For example, $G$ consists of 3 relations: \texttt{car} contains the model, make, door number and base price of a car, \texttt{brand} provides the headquarters’ location of a car manufacturer and \texttt{ad} stores the price at which somebody sells a model.

The registration $R_i$ of each source $i$ consists by a (for simplicity) single mapping constraint ($U_i \subseteq V_i$) over $S_i$ and $G$, printed in Figures 3.2a-ii and 3.2b-ii and represented in 3.2a-i and 3.2b-i as a set of solid lines and arrows. The atoms of query $U_i$ (respectively $V_i$) are all relations of $S_i(G)$, that contain an attribute adjacent to an arrow or solid line. The equalities in $U_i$ ($V_i$) are depicted as solid lines between attributes in $S_i(G)$ and the distinguished variables of $U_i$ ($V_i$) correspond to attributes serving as arrow sources (targets). For example, source 1 provides ad prices, base prices and model descriptions for ads having the same ad and base price. Similarly source 2 provides the model, make and number of doors of some cars.

Note that differences between the source and global schema can be easily handled by the mapping language. For instance, by equating the values of \texttt{Model} in relations \texttt{car} and \texttt{ad}, ($U_1 \subseteq V_1$) specifies that the mapped ad and base price correspond to the same model. Similarly ($U_2 \subseteq V_2$) maps only source values that appear in pairs of tuples of \texttt{auto} and \texttt{detail} with the same \texttt{Id} (shown in Figure 3.2b-ii through a solid line between the 2 occurrences of \texttt{Id}).

\textbf{In the absence of target constraints.} Assuming that $\Delta_G = \emptyset$, we illus-
trate the 4 levels of self-reliance by presenting the contribution of the registration $R_2$ to the answer of 4 queries.

**Example 1. Self Sufficient**
Consider a query $Q_1$ asking for all car models manufactured in both a 2-door and a 4-door version:

$$Q_1(M) :\neg \text{car}(M, C_1, '2', B_1), \text{car}(M, C_2, '4', B_2).$$

$R_2$ is self sufficient w.r.t. $Q_1$, since, if source 2 is the only registered source, we can find some extent $DB_2$ of it, for which the set of certain answers to $Q_1$ is nonempty. Such an instance is the following:

$$DB_2 : \text{auto}(1, 'M3 98', 'BMW'), \text{detail}(1, '2', 'V6'), \text{detail}(1, '4', 'V6').$$

In this case all global instances satisfying mapping constraints $M_2$ contain tuples of the form $\text{car}(M, C_2, '2', X_1)$ and $\text{car}(M, C_2, '4', X_2)$ where $X_1, X_2$ are constants (potentially different among global instances). Thus the tuple ('M3 98') is an answer to $Q_1$ against each global instance consistent with $M_2$ (i.e. it is a certain answer).

**Example 2. Now Complementary**
Consider now another query $Q_2$ asking for the make and corresponding ad prices of cars:

$$Q_2(C, P) :\neg \text{car}(M, C, D, B), \text{ad}(M, S, P).$$

$R_2$ is now complementary w.r.t. $Q_2$ and $R_1$ for the following two reasons: 
First, $R_2$ is not self sufficient w.r.t. $Q_2$, since if only source 2 is present in the system, it does not provide any certain answers to $Q_2$ regardless of its extent. This follows from the fact that always one of the global instances satisfying $M_2$ will have an empty relation $\text{ad}$ (since $M_2$ does not place any restrictions on the contents of $\text{ad}$) and thus will give the empty answer to $Q_2$. Second, if both sources
1 and 2 are registered, $R_2$ contributes certain answers to $Q_2$ for some extents $DB_1$, $DB_2$ of the sources 1 and 2, respectively:

$DB_1 : post('M3 98', 'A. Brown', '25K', '25K')$
$DB_2 : auto('1', 'M3 98', 'BMW'), detail('1', '2', 'V6').$

In particular, the set of certain answers to $Q_2$ in presence of both $R_1$ and $R_2$ equals \{('BMW', '25K')\}, whereas in presence of $R_1$ alone it is empty (since $M_1$ does not restrict the values of $\text{Carmake}$).

\[\square\]

**Example 3. Later Complementary**

Assume now that we add an atom to $Q_2$. The new query $Q_3$ returns the make and ad prices of models but only if there exist data about their manufacturer:

$$Q_3(C, P) : - \text{car}(M, C, D, B), \text{ad}(M, S, P), \text{brand}(C, O).$$

Since none of $M_1$, $M_2$ restricts the instances of target relation $\text{brand}$, the set of certain answers is empty both in the presence of $R_1$ alone and $R_1$, $R_2$ together. Thus $R_2$ is neither self sufficient, nor now complementary w.r.t. $Q_3$ and $R_1$.

However $R_2$ is later complementary, since there exists a new registration $R_3$ s.t. $R_2$ is now complementary w.r.t. $Q_3$ and $\overline{R} = R_1, R_3$. $R_3$ registers a source with schema $S_3 = \{\text{manufacturer}(\text{Carmake}, \text{Origin})\}$, which provides the headquarter’s location for car manufacturers: $R_3 = (S_3, M_3)$ with $M_3 = \{(U_3 \subseteq V_3)\}$ and $U_3(C, O) : - \text{manufacturer}(C, O), V_3(C, O) : - \text{brand}(C, O)$. Indeed for some extents $DB_1, DB_2, DB_3$ of sources 1 through 3, respectively, $R_2$ contributes to the certain answers of $Q$:

$DB_2 : auto('1', 'M3 98', 'BMW'), detail('1', '2', 'V6')$  
$DB_3 : \text{manufacturer}(\text{BMW}, \text{Germany})$.  

In this case the set of certain answers to $Q$ in presence of all sources is \{('M3 98', '25K')\}, while in the absence of source 2, it is empty (since $R_1$ and $R_3$ do not restrict the value of $\text{Carmake}$ in the $\text{car}$ tuples).  

\[\square\]
Example 4. Unusable
Consider using again $Q_2$ but projecting also the base price (i.e. return the make, base and ad price of cars):

$$Q_4(C, B, P) :- car(M, C, D, B), ad(M, S, P).$$

$R_2$ is unusable w.r.t. $Q_4$, because, regardless of any sources registered in the future and of the extents for them and for source 1, $R_2$ cannot contribute certain answers to $Q_4$. Indeed, consider arbitrary future registrations $\overline{R}$ and extents for both the new sources and source 1. By definition of certain answers, for any tuple $(C, B, P)$ that is a certain answer to $Q_4$, there will exist a pair of tuples of the form $car(X_1, C, X_2, B), ad(X_1, X_3, P)$ in all global instances satisfying $R_1$ and $\overline{R}$ (where $X_i$, $1 \leq i \leq 3$ are constants possibly different among global instances). However the existence of these tuples is not imposed by $M_2$, since it does not restrict, either the ad tuples, or the value of the Baseprice attribute of the car tuples contained in the global instances. Thus by removing the registration $R_2$ from the system, we will get the exact same set of certain answers to $Q_4$. This means that $R_2$ does not contribute to this set and so it is unusable w.r.t. $Q_4$. \hfill \blacksquare

In the presence of target constraints. In the presence of integrity constraints on $G$ (i.e. if $\Delta_G \neq \emptyset$), the contribution of an individual source is harder to determine, because of the interference via integrity constraints with data provided from other sources. Since each constraint restricts the set of possible global instances, it may change the set of certain answers to a query and consequently the contribution of a source to it. We illustrate this fact by presenting the effect of adding target constraints on some of the previous examples.

By adding constraints on $G$, an application query $Q$ may become unsatisfiable. In this case, the set of certain answers to $Q$ is always empty and hence every registration becomes unusable w.r.t. it. Such a case is described in Example 5.

Example 5.
Consider again query $Q_1$ from Example 1, asking for models produced both with
2 and 4 doors. If we add the target constraint:

\[ \delta_1 : (U'_1 \subseteq V'_1) \text{ with} \]
\[
U'_1(M, C_1, D_1, B_1, C_2, D_2, B_2) :- \]
\[ car(M, C_1, D_1, B_1), car(M, C_2, D_2, B_2) \]
\[ V'_1(M, C, D, B, C, D, B) :- car(M, C, D, B) \]

stating that Model is the primary key (PK) of car, then \( Q_1 \) becomes unsatisfiable and thus \( R_2 \) (which is self sufficient w.r.t. \( Q_1 \) when \( \Delta_G = \emptyset \)) becomes unusable. □

However in other cases the addition of a target constraint may instead increase the self-reliance of a registration. Example 6 illustrates how by adding a primary key on the global schema an otherwise unusable registration becomes now complementary.

**Example 6.**

Look again at \( Q_4 \) asking for the make, base and ad price of models. \( R_2 \) (unusable w.r.t. \( Q_4 \) when \( \Delta_G = \emptyset \)) advances to now complementary w.r.t. \( Q_4 \) and \( R_1 \) if Model is a PK of car (i.e. if \( \Delta_G = \{\delta_1\} \)). Indeed, \( R_2 \) contributes certain answers to \( Q_4 \) for the source extents shown in Example 2. In particular, all global instances consistent with \( R_1 \) and \( R_2 \) contain tuples \( \text{car}(\text{M3 98'}, X_1, X_2, '25K') \), \( \text{ad}(\text{M3 98'}, X_3, '25K') \) and \( \text{car}(\text{M3 98'}, \text{BMW}', '2', X_4) \) \((X_i \text{ may be different across instances})\) and in those satisfying also \( \Delta_G \), the two partially specified car tuples are merged into \( \text{car}(\text{M3 98'}, \text{BMW}', '2', '25K') \). Thus in presence of \( R_1 \) and \( R_2 \), the set of certain answers to \( Q_4 \) contains ('BMW', '25K', '25K'), while if \( R_1 \) is registered alone within the system it is empty. Intuitively, a PK on \( G \) allows a source to contribute to a query \( Q \) by providing only part of a tuple required by \( Q \) (the values of the other required attributes of the tuple can be obtained by another source via the PK). □

Other types of target constraints may also have the same effect. Example 7 shows how the addition of referential integrity constraints on \( G \) can lead to the self-reliance increase of some registration.
Example 7.
Consider again $Q_3$ asking for the make and ad prices of models for which there exists info about their manufacturer. If we declare the referential integrity constraint:

$$\delta_2 : U'_2 \subseteq V'_2 \text{ with } U'_2(C) : - \text{car}(M, C, D, B)$$
$$V'_2(C) : - \text{brand}(C, O)$$

which states that for each car tuple, a brand tuple with the same make also exists, then each global instance consistent with $M_1$, $M_2$ and $\delta_2$ contains for each car tuple provided by $M_2$ a corresponding brand tuple. Hence $R_2$ (which is later complementary in the absence of target constraints) becomes now complementary w.r.t. $Q_3$ and $R_1$ when $\Delta_{G} = \{ \delta_2 \}$.

3.5 Main Results

We study only self-sufficiency, now-complementarity and later-complementarity, as unusability is by definition reducible to them.

The decidability of the self-sufficient and now-complementary tests is based on reducing them to reasoning about the certain rewriting of the application query $Q$, in particular to checking satisfiability, respectively containment of certain rewritings.

**Certain Rewritings.** Given registrations $\{R_i = (S_i, M_i)\}_{1 \leq i \leq n}$, a certain rewriting of an application query $Q$ against $G$ is a query against the combined schemas $S_i$, which returns precisely the certain answers to $Q$ for any source extents $\overline{DB}$. Specifically, denoting a certain rewriting with $\text{rewr}_{\overline{\Pi}}(Q)$, we have:

$$\text{rewr}_{\overline{\Pi}}(Q)(\overline{DB}) = \text{cert}_{\overline{DB}}^{M}(Q).$$

Example 8.
Recall $Q_1$ from Example 1, asking for models with both 2 and 4-door versions. Its certain rewriting in $R_2$’s presence is:
\[ Q'_1(M) : auto(I_1, M, C_1), detail(I_1, 2', E_1), \]
\[ auto(I_2, M, C_2), detail(I_2, 4', E_2). \]

Note that it is not a priori clear that any certain rewriting exists, and even if it does, it is not given that it is expressible in a language in which containment and satisfiability are decidable. Indeed, even for pure LAV data integration scenarios (which are subsumed by the GLAV systems considered here), in the presence of functional dependencies (FDs) on the global schema, the solutions proposed in the literature [DGL00] yield rewritings expressed as recursive Datalog programs, for which containment is undecidable [AHV95].

However, in Theorem 12 in Section 3.5.4, we show that when there is only at most one FD per relation in \( G \), the certain rewriting can be obtained as a union of conjunctive queries, for which containment is in \( NP \). For the sake of presentation, we present this result last. Notice that the case of allowing only primary key constraints is covered by this result, since PKs are particular cases of FDs, and there can be at most one PK per relation.

When not specified otherwise, in the following results all registrations are given by sets of mapping constraints from IC\( (CQ^-) \), and all queries belong to UCQ\( ^- \).

### 3.5.1 Self-Sufficient Registrations

Our first result concerns the decidability of testing whether a registration \( R \) is self-sufficient when the global schema contains only primary keys.

This result is based on the following:

**Lemma 1.** Assume that \( Q \) has a certain rewriting \( \text{rew}_R(Q) \) expressed in some query language. Then \( \text{Suf}_Q(R) \) holds if and only if \( \text{rew}_R(Q) \) is satisfiable.

Lemma 1 and Theorem 12 imply:

**Theorem 5.** If \( \Delta_g \) contains only primary keys, then for any application query \( Q \in UCQ^- \) it can be decided whether \( \text{Suf}_Q(R) \) holds in PTIME in the size of \( \text{rew}_R(Q) \).
Example 9. The result presented in Example 1 that $R_2$ is self-sufficient w.r.t. $Q_1$ can be inferred by checking that $rew_{R_2}(Q_1)$, shown in Example 8, is satisfiable. □

Theorem 5 does not apply in the presence of RICs because the results in [DGL00] do not provide a certain rewriting in that case. We actually obtain the following result:

**Theorem 6.** It is undecidable to check that, given $\Delta_g$ containing PKs and RICs, registration $R$ and $Q \in CQ^-$, $Suf_Q(R)$ holds.

The proof is by a reduction from the Post Correspondence Problem [AHV95].

Since both PKs and RICs are expressible in IC($CQ^-$), we immediately obtain the following corollary:

**Corollary 1.** It is undecidable to check that, given $\Delta_g \subseteq IC(CQ^-)$, registration $R$ and $Q \in CQ^-$, $Suf_Q(R)$ holds.

### 3.5.2 Now-Complementary Registrations

The basis for our decidability result for checking now-complementarity consists in a reduction to query containment. We first state the reduction for the most general case it applies in, although for the purpose of decidability we will have to consider only primary keys on the global schema.

The reduction holds when full dependencies are allowed on the global schema. Full dependencies are IC($CQ^-$) queries of the form $(U \subseteq V)$, where all of $V$’s variables appear in its head. Full dependencies include all primary keys, functional dependencies, and restricted referential integrity constraints in which $V$ does not project away any attributes.

**Theorem 7.** Given $\Delta_g$ containing full dependencies and given registrations $\overline{R}, R_{n+1}$, we have that $NComp^Q_{\overline{R}}(R_{n+1})$ holds if and only if

(i) $rew_{R_{n+1}}(Q)$ is not satisfiable and
(ii) $rew_{\overline{R}, R_{n+1}}(Q)$ is not contained in $rew_{\overline{R}}(Q)$.

Theorem 7 and Theorem 12 yield the following:
Corollary 2. If $\Delta_G$ contains only PKs, then $N\text{Comp}^Q_{\overline{\Pi}}(R_{n+1})$ can be checked in $\text{NP}$ in the size of $\text{rewr}_{\overline{\Pi}}(Q)$ and in $\text{PTIME}$ in the size of $\text{rewr}_{\overline{\Pi},R_{n+1}}(Q)$.

We next expose an interesting connection between the problem of deciding self-sufficiency and that of deciding now-complementarity. This will help us transfer undecidability results from the former to the latter.

Theorem 8. The problem of deciding self-sufficiency of a registration is reducible to that of deciding now-complementarity.

Theorem 6 and Theorem 8 yield:

Corollary 3. It is undecidable to check, given $\Delta_G$ containing PKs and RICs, query $Q \in \text{CQ}^=$ and registrations $\overline{R}, R_{n+1}$, whether $N\text{Comp}^Q_{\overline{\Pi}}(R_{n+1})$ holds.

Corollary 3 implies undecidability in the presence of IC($\text{CQ}^=$) integrity constraints on $G$.

### 3.5.3 Later-Complementary Registrations

The inherent difficulty in deciding later-complementarity lies in having to check now-complementarity with any one of an infinite set of possible registrations. We first show that, in the absence of target constraints ($\Delta_G = \emptyset$), we can confine the test to a canonically chosen registration, defined below:

**Identity Registration.** Given $G$, the registration consisting of schema $G$ and mappings copying its extent to the global database is called the identity registration $R_{id}$:

$$R_{id} = (G, M) \text{ with } M = \{ (U_i \subseteq V_i) : 1 \leq i \leq n \} \text{ and } U_i(\bar{x}_i) \vdash r_i(\bar{x}_i),$$

$$V_i(\bar{x}_i) \vdash r_i(\bar{x}_i),$$

where $r_i$ the relations of $G$ and $\bar{x}_i$ their attributes ($1 \leq i \leq n$).

If there are no constraints on $G$, later-complementarity reduces to now-complementarity w.r.t. $R_{id}$:

Theorem 9. Assume that $\Delta_G = \emptyset$ and $R_{n+1}$ is not now complementary w.r.t. $Q$ and $\overline{R}$. Then $L\text{Comp}^Q_{\overline{\Pi}}(R_{n+1})$ holds iff $N\text{Comp}^Q_{R_{id}}(R_{n+1})$ holds.
Corollary 2 and Theorem 9 yield the following:

**Corollary 4.** If $\Delta_G = \emptyset$, whether $LComp^Q_R(R_{n+1})$ holds can be decided in NP in the size of $rew_{\pi}(Q)$ and of $rew_{R_{id}}(Q)$ and in PTIME in the size of $rew_{\pi,R_{n+1}}(Q)$ and of $rew_{R_{id},R_{n+1}}(Q)$.

Example 10 below shows that the test based on the identity registration fails in the presence of integrity constraints and Theorem 9 cannot be relaxed to allow even PKs in $\Delta_G$.

**Example 10.** Consider a simplified version of our running example. In an initially empty system we are adding source 1 via registration $R_1$ shown in Figure 3.3a. The source provides Audi models and their base prices (the equality atom $\text{Carmake} = 'Audi'$ is depicted as a box with the value 'Audi' next to $\text{Carmake}$). Assuming that $\text{Model}$ is the PK of $\text{car}$, we want to check the contribution of $R_1$ to a query asking for the base price of BMW M3 98: $Q(B) :- \text{car}(M3\ 98, 'BMW', B)$. $R_1$ is obviously neither self sufficient, nor now complementary w.r.t. $Q$, since it provides only data about Audis, while the query asks for the price of a BMW.
We check whether \( R_1 \) is now complementary w.r.t. \( Q \) and \( R_{id} \) (shown in Figure 3.3b). In order to get a certain answer to \( Q \) in presence of \( R_{id} \) and \( R_1 \), a tuple for M3 98 has to be provided by the source with registration \( R_{id} \) (it cannot be given by source 1, because it would imply that the make of M3 98 is Audi and thus we would not get any certain answer to \( Q \)). But \( R_{id} \) by definition, when giving the tuple of M3 98, will also specify all other attributes (i.e. make and price) of the tuple for which \( Q \) is asking and thus it will not allow the first source to contribute anything to \( Q \). Hence \( N\text{Comp}_{R_{id}}^Q(R_1) \) does not hold.

However \( R_1 \) can contribute to \( Q \) if source 2 with registration \( R' = (S', M') \) joins the system, where its schema \( S' \) consists of a single relation \( \text{pair}(\text{Model}_1, \text{Model}_2) \) and its set of mappings is \( M' = \{ (U' \subseteq V') \} \) with
\[
U'(M_1, M_2) : \text{pair}(M_1, M_2) \text{ and } \\
V'(M_1, M_2) : \text{car}(M_1, 'BMW', B), \text{car}(M_2, C, B),
\]
which means that source 2 provides pairs of BMW models with other models of the same price. Indeed, assuming the extents: \( \text{DB}_1 : \text{auto}('S4 97', '25K') \) and \( DB_2 : \text{pair}('M3 98', 'S4 97') \) for sources 1 and 2, respectively, \( R' \) alone does not provide any certain answers to \( Q \) but \( R_1 \) and \( R' \) together do. Intuitively \( R_1 \) contributes the price of BMW M3 98 by providing the price of Audi S4 97 (which according to the data given by source 2 is equal to the base price of BMW M3 98). Hence \( R_1 \) is later complementary but not now complementary w.r.t. \( Q \) and \( R_{id} \).

Example 10 also shows that the search for the future registration \( R' \) has to consider mappings whose queries perform self-joins. This poses a problem, since even if we picked one of the infinitely many schemas for \( S' \), we’d still be left with an infinite search space obtained by considering all CQ\(^= \) queries over \( G \) for the conclusion of the mapping constraints in \( M' \) (not to mention the infinite search space for the premise). Our decidability result is based on bounding these search spaces as follows:

Let \( R = (S, M) \) be the registration of our source, \( Q \) the application query, and let \( \text{size}(Q) \) be the sum of the arities of all relational atoms in \( Q \).

**Theorem 10.** Assume that \( N\text{Comp}_{\pi}^Q(R) \) does not hold, and \( \Delta_g \) consists only of PKs, with \( m \) denoting the maximum size of a key.
Then $\text{LComp}_R^Q(R)$ holds iff $\text{NComp}_R^Q(R)$ holds, where $R' = (S', M')$ and $M' = \{(U' \subseteq V')\}$ such that for some $1 \leq k \leq \text{size}(Q)$, we have

- $S'$ contains a single relation $D$ of arity $2 \times m \times k$, and
- $U'$ is a projection of $D$, and
- $V'$ is a query over $G$, of arity at most $2 \times m \times k$, with at most $2k$ atoms, and whose constants (if any) appear in $M$ or $Q$.

There are therefore exponentially many (in the size of $Q$) candidate registrations to consider. However, each is of size polynomial in that of $Q$.

Example 11. For the registration $R_1$ from Example 10, we have $m = 1$ and $\text{size}(Q) = 3$. Registration $R'$ is found for $k = 1$: $D$ is the relation pair of arity $2 \times 1 \times 1$, $V'$ has arity 2 and has 2 atoms, while $U'$ is a projection of pair. \qed

3.5.4 Non-recursive Certain Rewritings

Previous work [AD98, DGL00] describes only algorithms which yield a certain rewriting as a recursive Datalog program even in local-as-view scenarios, as soon as there are functional dependencies (FDs) in the target schema. This is justified by an example integration scenario in [DGL00] where the FDs force the certain rewriting to be recursive. However, the example uses several FDs per relation. We show here (Theorem 12 below) that, if each relation in the target schema allows at most one FD, then there exists an equivalent, nonrecursive rewriting expressed as a union of conjunctive queries with equalities (UCQ\textsuperscript{=}). Moreover, we show that such a rewriting can be constructed by reusing an algorithm developed in previous work for exact rewritings under the closed world assumption [DT03b, DT05]. These results are of independent interest as they extend previous work on finding certain rewritings, and they shed additional light on the connection between exact rewritings in the closed world assumption and certain rewritings in the open world assumption.

Exact Rewritings. Given registrations $\{R_i = (S_i, M_i)\}_{1 \leq i \leq n}$, an exact rewriting of an application query $Q$ against $G$ is a query $xrew_{\text{R}}^G(Q)$ against the
combined schemas $\mathcal{S}_i^r$, which returns precisely the answer to $Q$ for any source extents $DB$:

$$\forall DB \forall (G \in Targets_{\mathcal{M}}(DB)) \ xrewr_{\mathcal{M}}(Q)(DB) = Q(G).$$

Notice that any exact rewriting is also a certain rewriting. Clearly, in the classical GLAV scenario in which all mappings are given by source-to-target inclusion constraints [FKMP03] (this is the case in all our examples as well), there is no exact rewriting as the inclusions between source data and target data are not violated by adding tuples to the target. This changes $Q(G)$ but does not affect any query over the sources, so its answer cannot be the same with $Q(G)$ for all targets $G$. Exact rewritings are therefore meaningful only under the Closed World Assumption.

**Closed World Assumption.** Traditionally, the Open and Closed World Assumptions (OWA, respectively CWA) are defined only when the registration mappings are given by views [AD98]. We extend the definitions to constraints. Given source extents $DB$, registration mappings $\mathcal{M}$ and target $G \in Targets_{\mathcal{M}}(DB)$, we say that $G$ is a *minimal target* of $DB$ under $\mathcal{M}$ if no proper sub-instance $G'$ of $G$ is also a target ($G' \notin Targets_{\mathcal{M}}(DB)$). We say that the registration mappings $\mathcal{M}$ satisfy the Closed World Assumption (in short, $\mathcal{M}$ are CWA mappings) if for any source extents, all targets under $\mathcal{M}$ are minimal. Otherwise, $\mathcal{M}$ are OWA mappings.

When the registration is given only by source-to-target inclusion constraints, the mappings are OWA.

Previous work [DT03b, DT05] has shown that the Chase&Backchase algorithm introduced in [DPT99] is sound and complete for finding all minimized exact rewritings when the query is expressed in UCQ$^=$ and the constraints in IC(UCQ$^=$). Ignoring minimization, the results in [DT03b, DT05] imply the existence of the following algorithm, which is guaranteed to construct a canonical exact rewriting whenever it exists (according to Theorem 11 below).

Procedure **CanRewr**$_{\mathcal{S},G,\Delta_G,\mathcal{M}}(Q)$
1. chase \( Q \) with \( M \cup \Delta_G \) to obtain a query \( U \in \text{UCQ}^- \) formulated against the combined schemas \( \mathcal{S}, \mathcal{G} \). For an extension of the chase to \( \text{UCQ}^- \) queries and \( \text{IC(UCQ}^-) \) constraints, see [DT05, DLN05].

2. construct a query \( U|_{\mathcal{S}} \) by dropping from all of \( U \)'s rules all atoms over \( \mathcal{G} \). Drop all the rules that have become unsafe (this might result in the empty set of rules, corresponding to the unsatisfiable query).

3. return \( U|_{\mathcal{S}} \).

A common scenario requiring exact rewriting under the closed world assumption is that of rewriting using materialized views.

**Example 12.** Exact rewriting using views.
Consider the query \( Q(y, z) : - R(x, y, z) \) and the two materialized views \( V_1(x, y) : - R(x, y, z) \) and \( V_2(x, z) : - R(x, y, z) \). Also, let the first component of \( R \) be a key. Rewriting \( Q \) using only the views can be seen as finding an exact rewriting of \( Q \) using the following CWA mapping. The global schema is \( \mathcal{G} = \{ R \} \), the source schema is \( \mathcal{S} = \{ V_1, V_2 \} \) and the set of global target constraints \( \Delta_G \) contains only the key on \( R \). The mapping \( M \) consists of the constraints
\[
\{ V_1(x, y) \subseteq \exists z R(x, y, z), V_2(x, z) \subseteq \exists y R(x, y, z), \\
\exists z R(x, y, z) \subseteq V_1(x, y), \exists y R(x, y, z) \subseteq V_2(x, z) \}.
\]
Notice that the first and third constraints capture view \( V_1 \), stating the inclusions between its materialized extent and the defining query. Similarly, the second and fourth constraint capture view \( V_2 \). The first two constraints in \( M \) are source-to-target constraints (the queries are expressed as formulae with free variables), whereas the last two state inclusions from the target to the source database. It is easy to see that \( M \) satisfies the CWA, and that the query \( Q_r(y, z) : - V_1(x, y), V_2(x, z) \) is an exact rewriting of \( Q \). \( Q_r \) is constructed by algorithm \textsc{CanRewr} as follows. Chasing \( Q \) with \( M \cup \Delta_G \) yields
\[
U(y, z) : - R(x, y, z), V_1(x, y), V_2(x, z)
\]
(chase steps apply only with the last two constraints in \( M \)) and \( U|_{\mathcal{S}} \) yields \( Q_r \). \( Q_r \) turns out to be an exact rewriting, i.e. equivalent to \( Q \), as can be checked by
chasing $Q_r$ with the first two constraints in $M$ and with the key constraint, to find a containment mapping from $Q$ into the latter chase result.

If no exact rewriting exists, then algorithm CANREWR returns a \textit{minimally-containing} UCQ$^\omega$ rewriting of $Q$.

\textbf{Minimally-Containing Rewritings.} Given registrations $\{R_i = (S_i, M_i)\}_{1 \leq i \leq n}$, a containing rewriting of an application query $Q$ against $\mathcal{G}$ is a query $Q_c$ against the combined schemas $S_i$, such that:

$$\forall DB \forall (G \in \text{Targets}_{\overline{M}}(DB)) \; Q(G) \subseteq Q_c(DB).$$

$Q_m$ is a minimally-containing UCQ$^\omega$ rewriting of $Q$ if it is a containing rewriting and for any other containing UCQ$^\omega$ rewriting $Q_c$ of $Q$, we have

$$\forall DB \forall (G \in \text{Targets}_{\overline{M}}(DB)) \; Q_m(DB) \subseteq Q_c(DB).$$

\textbf{Theorem 11.} Assume that the chase of $Q$ with $\overline{M} \cup \Delta_G$ terminates, and that $Q$ has a containing UCQ$^\omega$ rewriting.

Then $\text{CANREWR}_{\overline{M} \cup \Delta_G, \overline{M}}(Q)$ is guaranteed to return a minimally containing UCQ$^\omega$ rewriting $Q_m$ of $Q$. In particular, if $Q$ admits an exact UCQ$^\omega$ rewriting, then $Q_m$ is guaranteed to be exact.

The proof was given in [DLN05]. It follows from the results in [DT03b, DT05].

\textbf{Finding certain rewritings.} The algorithm for finding certain rewritings in the OWA is based on modifying the original registration mappings $\overline{M}$ to obtain the CWA mappings $\overline{M}_f$, and then applying algorithm CANREWR on the new mappings. Its result will be the certain rewriting.

To obtain $\overline{M}_f$, we start from the idea introduced in the “inverse-rule algorithm” presented in [DGL00]. This algorithm turns a mapping given by a set $\overline{M}$ of IC(UCQ$^\omega$) constraints into one given by a \textit{logic program} $LP$ with function symbols, called Skolem functions. $LP$ has two properties: (i) it satisfies the CWA since its output is defined by the minimal model semantics, and (ii) $\text{cert}_{\overline{M}}(Q)$ can be computed by running $Q$ on $LP(\overline{DB})$ and removing from the result all tuples containing Skolem function terms.
Example 13. Inverse rules (adapted from [DGL00])
Consider the source schema $S = \{s\}$, the global schema $G = \{g\}$, and the mapping given by $M = \{(U \subseteq V)\}$ with $U(P,C) : - s(P,C)$ and $V(P,C) : - g(A,P,C)$, mapping source relation $s$ into the projection of global relation $g$. $M$ is turned into the logic program rule

$$g(Sk_F(P,C),P,C) : - s(P,C) \quad (3.1)$$

where $Sk_F$ is a fresh Skolem function symbol. By running this program on the extent of $s$, we get an extent for $g$ whose first column contains fresh invented values computed using $Sk_F$. If we fix $Sk_F$, the extent of $g$ is uniquely determined by that of $s$. An important requirement of $Sk_F$ (coming from [DGL00]) is that it never assign the same value to distinct arguments, unless forced to do so by an FD in $\Delta_g$. For instance, if we assume the FD $fd_1 = C \rightarrow A$ on $g$, then $Sk_F(P,C) = Sk_F(P',C)$ for any values of $P, P'$, in order not to violate $fd_1$. However, $Sk_F(P,C) \neq Sk_F(P,C')$ for all $C \neq C'$. In the absence of any FDs, $Sk_F$ must be injective. We refer to this property of $Sk_F$ as injectivity modulo FDs. [DGL00] shows how $LP$ can be turned into a Datalog program $DP$ without function symbols, which is a certain rewriting. However, in the presence of FDs in $\Delta_g$, $DP$ is recursive and thus doesn’t serve our purpose. We take an alternate approach, turning $LP$ into a new set of mapping constraints $M_f$, together with a set of new target constraints $\Delta_g^f$. For simplicity of presentation, we only illustrate the construction of $M_f$ and $\Delta_g^f$ on the above example.

Example 14. Capturing inverse rules with constraints
We first obtain the new global schema $G_f$ by extending schema $G$ with a relation $F$ modeling the Skolem function (its intended meaning is that $F(A, P, C)$ iff $Sk_F(P,C) = A$). Then we eliminate the Skolem terms from the logic program rules, substituting $F$ for $Sk_F$. (3.1) thus becomes $(U_0 \subseteq V_0) \in M_f$ where

$U_0(P,C) : - s(P,C)$
$V_0(P,C) : - g(A,P,C), F(A,P,C)$. 
To ensure that $F$ models a function, we add to $\Delta_f^f$ the corresponding FD $PC \rightarrow A$ on $F$, expressible as $U_1 \subseteq V_1$:

$$U_1(A_1, A_2) : - F(A_1, P, C), F(A_2, P, C),$$
$$V_1(A, A) : - F(A, P, C).$$

We specify $F$’s injectivity modulo FDs as follows. Recall from Example 13 that in the presence of $fd_1$, $Sk_F$ will return the same $A$ value on arguments agreeing on the $C$ value. To express that this is the only case when the results of $Sk_F$ coincide, we add to $\Delta_f^f$ the constraint $(U_2 \subseteq V_2)$:

$$U_2(A, C, C') : - g(A, P, C), F(A, P, C),$$
$$g(A, P', C'), F(A, P', C'),$$

We next turn the mapping into one satisfying the CWA. Consider the decomposition $\rho = \{AC, PC\}$ of $g$. This is a lossless join decomposition, which cannot be further decomposed without compromising the lossless join property [AHV95]. For each $X \in \rho$, we introduce a corresponding constraint into $M_f$, stating that all tuples in $\pi_X(g)$ must stem from the source via the evaluation of the logic program rule on $s$ (this is the minimality requirement). Specifically, we add $\{U_3 \subseteq V_3, U_4 \subseteq V_4\}$ to $M_f$, where

$$U_3(A, C) : - g(A, P, C),$$
$$V_3(A, C) : - s(P', C), F(A, P', C),$$
$$U_4(P, C) : - g(A, P, C),$$
$$V_4(P, C) : - s(P, C).$$

Notice that these two constraints state inclusions from the target to the source.

Now consider the query $Q(P_1, P_2) : - g(A, P_1, C_1), g(A, P_2, C_2)$. The algorithm $\text{CanRewr}_{S, g_f, \Delta_f^f, M_f}(Q)$ proceeds as follows. It first chases $Q$ with $(U_4 \subseteq V_4)$ (two steps), then with $(U_0 \subseteq V_0)$ (two steps) and $(U_3 \subseteq V_3)$ (another two), followed
by two steps with the PK on $g$, and finally with $(U_2 \subseteq V_2)$ (one step) to obtain

$$U(P_1, P_2) :\text{-} g(A, P_1, C_1), g(A, P_2, C_2),$$
$$s(P_1, C_1), s(P_2, C_2),$$
$$g(A', P_1, C_1), F(A', P_1, C_1)$$
$$g(A'', P_2, C_2), F(A'', P_2, C_2)$$
$$s(P'_1, C_1), F(A, P'_1, C_1),$$
$$s(P'_2, C_2), F(A, P'_2, C_2),$$
$$A = A', A = A'', C_1 = C_2$$

$U|_S$ yields

$$Q_m(P_1, P_2) :\text{-} s(P_1, C_1), s(P_2, C_1), s(P'_1, C_1), s(P'_2, C_1),$$

which minimizes trivially to $Q_m(P_1, P_2) :\text{-} s(P_1, C_1), s(P_2, C_1)$. This is the certain rewriting of $Q$. Intuitively, this is because $Q$ requires $P_1, P_2$ to be associated with the same $A$. Since the value of $A$ is not provided by the source, we use the next best condition, namely we require $P_1, P_2$ to coincide on $C$. Since the latter determines $A$, all tuples returned by $Q_m$ are valid.

It turns out that, for several FDs per relation, the constraints for injectivity modulo FDs become more complex, leading to guaranteed non-termination of the chase. Interestingly, the successive queries obtained during the infinite chase sequence enumerate the unfoldings of the recursive Datalog program constructed in [DGL00] from $LP$.

Algorithm CREWR($\mathcal{S}, \mathcal{G}, \Delta_\mathcal{G}, \mathcal{T}, Q$)

1. Construct the “inverse-rules” logic program $LP$ corresponding to mapping $\overline{M}$.
2. Construct a new global schema $\mathcal{G}_f$, a new set of target constraints $\Delta'_\mathcal{G}$, and new mapping constraints $\overline{M}_f$ which induce the same mapping as $LP$.
3. Return $\text{CANREW}_{\mathcal{S}, \mathcal{G}_f, \Delta'_\mathcal{G}, \mathcal{T}_f}(Q)$.

**Theorem 12.** If $\Delta_\mathcal{G}$ contains at most one functional dependency per relation, then for any list of registrations $\mathcal{T}$ and any application query $Q \in \text{UCQ}^\infty$, $Q$ has a nonrecursive certain UCQ$^\infty$ rewriting and algorithm CREWR finds such a rewriting.
The crux of the proof is that (i) $\mathcal{M}_f$ induces the same mapping as the inverse-rule logic program $LP$, and (ii) the chase with the constraints in $\mathcal{M}_f \cup \Delta'_G$ is guaranteed to terminate for any application query $Q$, so that Theorem 11 applies.

3.6 Related Work

[DGL00] provides the “inverse-rule” algorithm for finding maximally contained rewritings for LAV integration, and [AD98] shows that maximally contained rewritings coincide with certain rewritings. The result was extended in [FLM99] to the GLAV case, but in the absence of integrity constraints on the global schema. A further extension to the nested relational and XML data models is provided in [YP04]. [HIM+04] shows how to find the certain rewritings in a GLAV-like setting used in peer mediation. [Koc02] settles the complexity and decidability of checking the existence of this rewriting (the GLAV mappings are called “symmetric constraints” there), considering both cyclic and acyclic sets of mapping constraints. In the presence of functional dependencies on the global schema, the inverse-rule algorithm returns a recursive Datalog program, defeating our goal of reasoning about containment of certain rewritings. [MHF02] decides containment of certain rewritings in LAV scenarios without integrity constraints. The authors consider limited access capabilities, which also result in certain rewritings expressed as recursive Datalog programs. [CCGL02] reduces a GLAV mapping into a pure GAV mapping, exploiting integrity constraints on the global schema. [CGL01] extends the ideas in [CCGL02] to allow for inclusion dependencies on the global schema. This work is extended in [CLR03], which performs a comprehensive study of the complexity of answering application queries for certain answers, under the Open World Assumption, but also for various relaxations of the query semantics which are appropriate for inconsistent and incomplete information. [CLR03] shows that query answering is undecidable in the presence of arbitrary PKs and inclusion dependencies and that, for restrictions on the inclusion dependencies, the problem becomes decidable. Various complexities are obtained for various classes of constraints and queries. Despite the fact that the work in [CLR03] assumes the
source instance given while our definitions are instance-independent, we plan to investigate whether the restrictions in [CLR03] lead to improved complexities in our setting.

3.7 Conclusions

Self-reliance levels model trade-offs between reliance of a source on other sources to provide to an application query and cleaning cost required to make the data conform to the global schema. In chapter 2 we presented the set of self-reliance levels used by RIDE and in this chapter we present an alternative set of definitions and study their decidability for different classes of target constraints.

This chapter is, in part, a reprint of material that appeared in PODS 2005 and was authored by Alin Deutsch, Yannis Katsis and Yannis Papakonstantinou.
Chapter 4

Inconsistency Resolution

This chapter describes Ricolla; a wiki-inspired online database that supports collaborative inconsistency resolution for communities. The chapter is structured as follows: Section 4.1 provides some necessary background. Section 4.2 contains an overview of the system and Section 4.3 describes the system’s architecture. Finally, Sections 4.4 - 4.8 describe the technical details of the system, Sections 4.9 and 4.10 present the related and future work, respectively and Section 4.11 concludes the chapter.

4.1 Background

The increasing need of communities to collaboratively maintain structured data led recently to the proliferation of web-applications that provide this functionality, referred to as online databases. However collaborative editing results in a major challenge: Since data originate from different users, they often contradict each other. These data conflicts usually arise either because of errors or due to differing opinions. Instances of the latter often appear in the sciences, where, as explained in [TI06], researchers have contradicting opinions, without being able to neither prove nor disprove them.

Current online databases treat data conflicts in ad-hoc and limited ways, captured by two extremes:

On one hand, online databases following the wiki paradigm, such as Dab-
ble DB [dab], blist [bli], QuickBase [qui] and Freebase [fre], *ignore the conflicts*. Being totally unconstrained, they allow users to enter any data even if they are contradicting. The advantage of this method is that it preserves all conflicting data for community members to see (in contrast to the second approach discussed next). However, it also severely restricts the utility of the stored data, as members are presented with conflicting data without an obvious way of highlighting inconsistencies and resolving them.

On the opposite end of the spectrum, traditional DBMS-driven online databases *disallow data conflicts* altogether. Enforcing a set of integrity constraints, they only allow data that satisfy the constraints. The obvious advantage is that community members see data that do not contradict each other. But this can also be a major drawback. The presence of conflicting data is often desirable, especially when they correspond to differing opinions, as discussed above.

It is thus obvious that current solutions do not meet the requirements of a large number of online communities for inconsistency management. Ideally community members should be able to both enter conflicting data (in contrast to DBMS-driven online databases) and have a way to easily inspect the inconsistencies and resolve them (in contrast to online database wikis). Moreover, to leverage the power of the community, they should be able to resolve the conflicts in collaboration with their peers, while being allowed to maintain different opinions when there is disagreement. Finally, the conflict resolution should happen in an “as-you-go” fashion, allowing members to use the system and query the data even before all conflicts have been resolved.

In this chapter we present **Ricolla** (*Resolve Inconsistencies in a COLLABorative environment*); a wiki-inspired online database, which by satisfying the aforementioned requirements, supports a natural workflow for the management of conflicting data. As we will explain next, Ricolla offers mechanisms that allow users to explicitly model conflicting data, inspect the conflicts and resolve them in an “as-you-go” fashion, while allowing for different opinions.

In the following we present the functionality and architecture of the system together with our contributions through a representative use-case: We want to
create an online movie database, where cinephiles can collaboratively edit information about movies. To bootstrap the system, the community initiator designs the schema of the database. The schema, shown in Figure 4.1, consists of 3 relations, holding information on movies, actors and their relationships.

4.2 System’s Overview

After the initiation of the system, individual movie fans can do the following:

Model and query conflicting data. Let us introduce Lara; a Clint Eastwood fan. Lara has recently discovered that her favorite movie actor is 1.85m tall and she wants to update the movie database to reflect that. By inspecting the database, she finds out that some other user has listed Clint’s height as 1.88m. In a DBMS-driven online database, she would be forced to replace 1.88m by 1.85m, since integrity constraints would allow only a single height value for each actor. However that would result in a system whose values are biased by Lara, who after all might be wrong about Clint’s height. In Ricolla on the other hand, instead of replacing existing data, she can simply augment them with her own (conflicting) opinions. Utilizing the system’s GUI, she can add as another possible height for Clint Eastwood, next to 1.88m, the value 1.85m. Figure 4.2 depicts the tuple summarizing the information for Clint Eastwood after Lara’s insertion as shown in Ricolla’s GUI. Such a tuple is called an alternative-capturing tuple (in short ac-tuple). As we will formally explain in Section 4.4, an ac-tuple captures different ac-alternatives for the attribute values. For instance in Figure 4.2 it shows the

1All values used in our running example are real values that can be found on different websites at the time of writing. For instance, www.celebheights.com lists Clint Eastwood’s height as 1.85m, while www.imdb.com lists it as 1.88m.
two possible alternatives for the height and two possible alternatives for the fan mailing addresses (which were entered previously by other users).

Apart from introducing (conflicting) data, Lara can also query them, even when they contain conflicts. For example, utilizing Ricolla’s visual query builder, Lara formulates a query asking for all actors with an address in Burbank and their movies. Assuming that the system contains the two movies for Clint Eastwood shown in Figure 4.4, the system returns the query result shown in Figure 4.6. The query result is shown in the same way as the base data and so Lara can quickly grasp the inconsistencies that affect her query. For instance, the query result exhibits the information about the two possible values for Clint’s height. By allowing users to query data before inconsistencies are resolved, Ricolla supports “as-you-go” conflict resolution.

Contributions. To enable the aforementioned functionality we make the following two contributions:

C1. A data model for representing conflicting data (Section 4.4), called ac-database, and a corresponding GUI. It has formal foundations, as an ac-database instance summarizes a set of possible worlds. Moreover, in contrast to other data models for conflicting data (a.k.a. uncertain data), it is designed with conflict resolution in a wiki-setting in mind and satisfies some unique requirements: it is compact, intuitive and supports actions to resolve the conflicts.

C2. A class of queries such that their answers (which are in general arbitrary sets of possible worlds) can be still represented in our data model (Section 4.6). This is known in the literature as the class of queries under which
the data model is closed. By identifying it, we provide guarantees on which query answers can be represented in our data model without information loss (i.e. without losing any correlation between the tuples that might exist in the query answer).

**Resolve conflicts at different granularities while maintaining different opinions.** After inserting Clint’s height and inspecting the data through a query, Lara decides to resolve some of the inconsistencies. She knows that out of the two mailing addresses listed for Clint in Figure 4.2, the correct one is the one in Burbank. She can enter this information into the system by simply marking this ac-alternative as right. This action, called a resolution action and carried out on the same GUI that shows the conflicting data, allows her to naturally reduce the number of conflicts. Note that a resolution action affects only the particular user’s view of the community database. For example, Harry, another movie fan, would still see both addresses and could mark a different address as right. This allows users to maintain differing opinions, which is a major requirement especially in the sciences, as discussed above.

After resolving the conflict on Clint’s address, Lara decides to resolve the conflicts in the height values of the actors. Given actors’ reputation of inflating their heights when reporting them to media, she decides to choose for every actor in the database the smallest among the heights listed. Instead however of going to each actor tuple and manually selecting the minimum height, she employs the resolution policy builder to write a resolution policy that automatically selects the minimum height for each actor. To capture most common use-cases, resolution policies can resolve conflicts based on conditions that might involve not only the actual data (such as the actor’s height above) but also annotations on them, such as their provenance or the timestamp of their insertion. For example, Lara can write a policy specifying that she trusts the heights provided by her friend Harry more than those provided by other users.

**Contributions.** To enable this conflict resolution functionality we make the following contributions:

**C3.** A set of resolution actions that allow community members to resolve in-
individual conflicts. Different members can maintain different opinions by resolving conflicts in different ways. Another notable feature of the resolution actions is that they can be carried out not only on the base data but also on query answers (Section 4.7). This allows community members to avoid resolving all conflicts and instead lazily resolve only those that affect queries in which they are interested.

C4. A resolution policy language that allows community members to write rules that resolve multiple conflicts at once based on certain criteria (Section 4.5.2). These criteria may be based not only on the values appearing in the database but also on annotations of those values (e.g. timestamp) or other users’ opinions.

4.3 System’s Architecture

Figure 4.3 depicts the architecture of the resulting system. The Frontend, shown on the top, allows users to visually formulate queries and resolution policies, inspect the data, insert new data and carry out resolution actions. These actions are supported by the Backend, shown on the bottom. To make the system as flexible as possible, we have opted for an architecture that allows each individual user to independently decide which data she wants to see and which users’ resolution actions (if any) she wants to take into account. Dark boxes with rounded corners represent functions while rectangles correspond to internal data structures. Whenever users insert data into the system through the Frontend, these are appended to an append-only community ac-database. In parallel metadata about the insertion (such as the user’s name and the timestamp of the insertion) are stored in a separate storage area, containing the user actions. Resolution actions carried out by the users are also recorded in the same area. Essentially the community ac-database and the user actions storage contain a description of all the relevant edit history of the system. Each user can subsequently write a resolution policy over these two storage areas to create her own view of the community ac-database (depicted in Figure 4.3 as User Ac-Database). By allowing resolution policies to
operate not only on the data but also on the user actions, each community member can individually decide whether she wants to maintain her own opinion or reuse resolution actions carried out by her peers. For instance, she can choose to trust her own resolution actions when it comes to resolving heights, but trust her friend for fan mailing addresses.

Contributions. Architecturally, our work makes the following contributions:

C5. A design for an end-to-end system and the associated workflow for managing conflicting data in a community setting. Although individual aspects of the system (such as the data model) have been addressed in previous work, one of the main contributions of this work is the presentation of an end-to-end system whose every aspect has been specifically tuned for data
conflict management and resolution. To the best of our knowledge this is the first system that allows community members to do all of the following: a) represent conflicting data in a formally defined data model, b) resolve only those conflicts that affect the answer of a specific query (to avoid resolving conflicts of no interest to them) and c) maintain their own opinions on how certain conflicts have to be resolved.

**C6. Algorithms** for implementing the system on top of a relational DBMS (Section 4.8). These include algorithms to store and retrieve an ac-database, to answer queries over an ac-database and to execute resolution policies.

### 4.4 Data Model

A conflict resolution system should allow users or applications to not only add data to the system but also to express the conflicts between them. For instance, users should be allowed to specify that two data items are mutually exclusive (i.e. if one exists, the other cannot exist), or that they always have to co-exist (i.e. they are either both true or both false). Therefore data cannot be represented in the flat relational database model. Instead the system needs a more expressive data model that captures not only the data that could exist in the system but also their relationship. As we will discuss in Section 4.4.4 researchers have already proposed a multitude of such models, commonly referred to as data models for uncertain data. However most models have been created as general-purpose representations and are not suitable for conflict resolution which places some unique requirements on the data model. In particular, a data model suitable for resolving conflicts in an interactive “as-you-go” manner should satisfy the following properties:

1. **Conflict-awareness**: As mentioned above, the data model should capture the conflicts between data items. However, since there are different types of conflicts (with the mutual exclusivity and the co-existence being just two examples), the data model should be expressive enough to capture the most commonly encountered types of conflicts.
2. **Intuitiveness:** Since the data model will be exhibited on a GUI that is going to be employed by humans, the conflicts should be represented in an easy to understand way. For example, deciding whether two data items are mutually exclusive or not should be obvious and it should not require parsing and understanding complex provenance formulas. Note that the requirement for intuitiveness conflicts with that for expressiveness, since capturing more conflicts is bound to make the model less intuitive. Therefore the data model should strike the right balance between expressive power and intuitiveness.

3. **Compactness:** Another requirement motivated by the use of the system by humans, is the need for a compact representation. To prevent information overload, the data model should summarize all available options instead of just laying them all out in a verbose way.

4. **Support for resolution actions:** Since the goal of the system is to allow users to eventually resolve the conflicts, the data model should be tuned towards resolution actions. By looking at the data it should become obvious which data are conflicting and how they can be resolved.

In this section we describe a data model for representing conflicting information whose design was guided by these desiderata. We start by describing the data model, then we define its semantics and the corresponding data manipulation language and finally we compare it to other previously proposed models for uncertain data. The discussion on how instances of our data model are actually stored in the system is left for the implementation section (Section 4.8).

### 4.4.1 Definition

Our data model is structured around the notion of an *alternative-capturing tuple* (in short *ac-tuple*); a special type of tuple that captures mutually exclusive information about the same subject.

**Ac-Tuple Structure.** Before formally defining the notion of an ac-tuple, let us first introduce it through an example. Figure 4.2 shows an ac-tuple summa-
rizing the conflicting information on Clint Eastwood, entered by Lara and other movie fans. It shows two possible heights (1.85m and 1.88m) and two possible fan mailing addresses (Burbank, 91522 and Carmel, 93921).

As demonstrated through the example, an ac-tuple can be vertically partitioned in a set of nested tables (3 in our case), which we call ac-fragments and cover part of the ac-tuple’s schema. Each row in an ac-fragment represents a possible assignment of values for the set of attributes in that ac-fragment and is therefore called an ac-alternative. For instance, the right-most ac-fragment, contains two ac-alternatives, one capturing the fact that Clint Eastwood’s fan mailing address might be in Burbank with zip code 91522 and the other expressing the possibility that it is in Carmel and has zip code 93921.

Note that our data model is flexible enough to allow us to either correlate or not correlate the attribute values in a tuple by creating ac-fragments with appropriate schemas: To correlate the values of two attributes (e.g. city and zip code) we only have to create a fragment that covers both attributes (as is the case in our example). On the other hand, if we want to express the fact that the possible values of an attribute are independent of the possible values of some other...
attributes, we can put them in different ac-fragments. For instance, by having in our example the height in a different fragment than the city we are expressing the fact that Clint Eastwood’s height can be either 1.85m or 1.88m regardless of his fan mailing address. It is important to note that we can decide how to correlate attribute values in a per-tuple basis. This means that each ac-tuple can have ac-fragments of different schemas.

**Correlating Ac-Tuples Through Coloring.** Until now we looked only at a single ac-tuple. Let us now move to an ac-relation comprised of a set of ac-tuples. The main question when considering sets of tuples is whether they are correlated or not. Can we reason about the conflicts represented by each ac-tuple in isolation or do we have to look at multiple tuples simultaneously?

It turns out that to express even simple query answers, different ac-tuples within the same ac-relation have to be correlated. In our data model such correlations are depicted by coloring ac-fragments of different ac-tuples with the same color, as exhibited in the following example.

Assume that our system contains 3 ac-relations; the Actor ac-relation with the single Clint Eastwood ac-tuple shown in Figure 4.2, the Movie ac-relation shown in Figure 4.4 containing two movies and the MovieActor ac-relation shown in Figure 4.5 specifying that Clint Eastwood has played in both movies. If we execute a natural join between these 3 ac-relations and put a selection on City with Burbank (to see only actors with a mailing address in Burbank and their movies), the system returns the query result shown in Figure 4.6.

![Figure 4.6: Ac-tuples corresponding to movies of actors with a fan mailing address in Burbank](image-url)
The query result contains two ac-tuples corresponding to the two Clint Eastwood's movies. Note that the ac-fragment containing the height is the same across both tuples and has the same color (yellow or the lighter shade of gray in B/W). Intuitively when two ac-fragments (across tuples) are identical and share the same color, they correspond to the exact same object. Therefore if a certain ac-alternative turns out to be true in one, the same ac-alternative will be true in the other. For instance, in our example if actor Clint corresponding to the first tuple has height 1.85m, the same will hold for the actor corresponding to the second tuple, since they are the same person. Please note that coloring fragments is more expressive than simply grouping tuples by a set of attributes. For instance, even though we can express the same information as in Figure 4.6 without colors by simply grouping the movies by the actor, this is not true in general. For example, if the database contained another actor that played in the movie “Million Dollar Baby”, grouping the movies by actor would generate two tuples for the particular movie (each appearing in a different group), which would have to be correlated through coloring.

The Empty Ac-Alternative. Finally, the data model can also express the fact that a tuple might not exist. This is accomplished by the empty ac-alternative, shown visually as an ac-alternative without any values for the ac-fragments attributes. For instance, in the query result showing movies of actors with an address in Burbank (Figure 4.6) the ac-fragment covering the city and zip code contains in both tuples among others an empty alternative. This indicates that these two ac-tuples either exist in the query result answer (if Clint Eastwood’s address is in Burbank) or they do not (if his address is not in Burbank but in Carmel).

Formal Definitions. Let us now formally define our data model. We proceed in a top-down fashion, defining first ac-databases and then components they are built from:

Definition 1. AC-DATABASE & AC-RELATION: An ac-database consists of a set
of ac-relations, which in turn are sets of ac-tuples. Similarly to flat relations, each ac-relation has also an associated schema, which is a set of attributes.

**Definition 2. AC-TUPLE:** An ac-tuple is a set of ac-fragments. Every ac-tuple belongs to an ac-relation and it has a schema which is identical to that of the relation.

**Definition 3. AC-FRAGMENT & AC-ALTERNATIVE:** An ac-fragment is a relation whose schema is a subset of the schema of the ac-tuple in which it appears and whose rows are called ac-alternatives. An ac-fragment can also contain a special empty row, called an 'empty' alternative. The schemas of all ac-fragments of an ac-tuple form a partition of the ac-tuple’s schema. Finally, each ac-fragment may be also associated with a color. Two ac-fragments can share the same color but only if they are identical (i.e. have the same schema and contain the same set of alternatives).

### 4.4.2 Semantics

To satisfy the conflict-awareness requirement, described above, our data model has a formal underpinning: Each ac-database (or ac-relation) represents a set of possible flat databases (or flat relations). Each such flat database (or relation) is referred to in the literature as a possible world. Intuitively a possible world of an ac-relation can be created by picking for every ac-fragment in the ac-tuples of the relation exactly one of its ac-alternatives. We refer to every such pick of ac-alternatives for a given ac-tuple as an interpretation of that ac-tuple, defined below:

**Definition 4. AC-TUPLE INTERPRETATIONS:** An interpretation of an ac-tuple is a mapping from each fragment of the tuple to a single alternative within that fragment. Empty ac-alternatives are treated in a special way: When a fragment is mapped to the empty alternative, then the ac-tuple is mapped to the empty tuple.

Per the definition above, the interpretation of an ac-tuple is a flat tuple (potentially empty). For example, the ac-tuple of Figure 4.2 has the four interpretations shown in Figure 4.7.
If fragments were not colored, then we could consider each ac-tuple separately to create the possible worlds represented by the ac-relation. If they are however colored, then decisions taken for a fragment with a certain color should be consistent across all ac-tuples in which a fragment with the same color appears. To this end, we define the notion of compatible ac-tuple interpretations.

**Definition 5. Compatible Ac-Tuple Interpretations:** Given two ac-tuples and one interpretation for each, we say that the two interpretations are compatible if they map each fragment of the ac-tuples with the same color to the same alternative.

Given the definition of compatible ac-tuple interpretations, we can now define the possible worlds represented by an ac-relation and an ac-database:

**Definition 6. Possible Worlds Of An Ac-Database:** An ac-relation represents all possible flat relations containing exactly one interpretation of every tuple in the ac-relation, such that the interpretations are pairwise compatible. Similarly, an ac-database represents all flat databases that can be constructed by taking for each ac-relation in the ac-database one of the possible relational instances that it represents.

From now on we will use $\text{PWorlds}(I)$ to denote the possible worlds represented by an ac-database or ac-relation $I$.

### 4.4.3 Ac-Data Manipulation Language

A community member can manipulate the data of an ac-database by carrying out actions directly on the interface displaying the data (denoted as “Data
Figure 4.8: ‘User Actions’ table for ‘1.85m’ ac-alternative in Clint Eastwood’s ac-tuple 

<table>
<thead>
<tr>
<th>User</th>
<th>Timestamp</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lara</td>
<td>1/3/2009 3:55pm</td>
<td>Insert</td>
</tr>
<tr>
<td>Harry</td>
<td>1/5/2009 1:00am</td>
<td>Delete</td>
</tr>
</tbody>
</table>

Employing the data viewer, users can augment the community ac-database by either inserting ac-alternatives into existing ac-tuples or by creating new ac-tuples. The data viewer represents these statements internally in the ac-Data Manipulation Language (in short ac-DML) before it ships them to the system’s Backend for execution.

Ac-DML statements. Ricolla supports DML statements corresponding to the following 2 actions:

- **Insert Ac-Tuple:** To insert facts about a new object, a new ac-tuple has to be created. Since ac-tuples within the same relation may have different ac-fragments, the fragment partition of the tuple has to be specified at the time of insertion (and it cannot be modified afterwards). Moreover when inserting a tuple, a single ac-alternative should be inserted in each fragment.

- **Insert Ac-Alternative:** Alternative facts about an object can be created by inserting ac-alternatives to an existing ac-tuple. This is accomplished by specifying the tuple and the ac-fragment in which to insert the ac-alternatives, as well as the values of the attributes for the new ac-alternative.

Although users can insert ac-tuples, ac-fragments and ac-alternatives, currently they cannot color the ac-fragments. However colors will still appear in the
query results to explain the correlation between ac-fragments. An example of such a case is the query answer shown in Figure 4.6. Although the users did not specify any colors, these still appear as part of the query answer.

‘User Actions’ table. Finally, in order to allow community members to resolve conflicts based on their own actions or those of their peers, whenever a user inserts an ac-alternative (either as part of a new tuple or in an existing tuple), the system records metadata about this action in a special ‘User Actions’ table attached to each alternative. This table stores at most one tuple for each community member containing the last action that the user carried out for that alternative together with the timestamp of the action. An action can be either an insertion action carried out through the above ac-DML statements or a resolution action, which we will describe in Section 4.5.1. Figure 4.8 displays the ‘User Actions’ table for the ac-alternative ‘1.85m’ in the Clint Eastwood ac-tuple, after two users performed actions on it. The first row of the table (denoted by ‘Insert’) shows that Lara first inserted the alternative in the system. The second row on the other hand (denoted by ‘Delete’) stores the fact that Harry later marked the alternative as wrong through a resolution action. We discuss resolution actions in detail in subsequent sections.

4.4.4 Comparison to other data models

The problem of representing sets of possible worlds has been the focus of an extensive amount of research for a very long time. Researchers have proposed many data models for representing sets of possible worlds with the most influential one being Lipski’s v-tables [IL84]. The problem has gained traction again recently under the umbrella of representing uncertain and incomplete data. This led to the introduction of new data models, such as Uncertainty-Lineage Databases (ULDBs) [BSH+08] (proposed in the context of the Trio system [ABS+06]) and World-Set Decompositions (WSDs) [AKO07a] and U-relations [AJKO08] (both designed in the context of the MayBMS system [AKO07b]). In parallel the same problem has been studied in the context of probabilistic databases [LLRS97, DS07].

However existing data models do not satisfy most of the requirements of our
setting, listed in Section 4.4. For instance, ULDBs, v-tables and U-relations are non-intuitive and lack support for resolution actions (violating thus requirements 2 and 4) as they use formulas to represent the relationships between individual tuples in the database. This makes it hard to decide whether two tuples are conflicting as it requires reasoning on the formulas. In contrast, in an ac-database conflicts are immediately obvious just by looking at it (they simply correspond to ac-alternatives). The same holds for the possible ways to resolve them as we will describe next. WSDs are also non-intuitive but for a different reason. Having been built with efficient query evaluation in mind, they split the data corresponding to a single tuple into several relations. This makes the model ideal as a storage representation but unsuitable for a user interface.

It seems that recent proposals of data models have placed a higher importance on the expressive power of the models than on their suitability for user interfaces. For instance ULDBs and WSDs are complete (i.e. they can represent all finite sets of possible worlds). On the other hand, we took a different approach by creating a data model that, while non-complete, is both expressive enough and intuitive. The need for non-complete but more intuitive data models has also recently been recognized in [SBHW06]. Finally, this comparison refers to data models suitable for Ricolla’s GUI. While existing data models are not suitable for the user interaction, they could still be used at the storage layer. For instance, we could use a ULDB as the Backend and maintain the ac-database for the Frontend.

4.5 Resolution Actions & Policies

Once the data are represented as an ac-database, the community members can resolve the conflicts. They can accomplish that in two ways, corresponding to different granularities: One way is to look at a single ac-tuple at a time and resolve the conflicts that pertain to it by carrying out resolution actions. This allows users to make fine-grained decisions on the ac-tuple level. However it can also be costly if the user wants to resolve conflicts in lots of ac-tuples according to some high-level criteria (e.g. if she wants to select the minimum height for each actor, as
discussed above). To this end, the system allows the user to express *resolution policies*; rules that summarize a set of resolution actions and apply them to many ac-tuples. Resolution actions and policies are described in the following sections.

4.5.1 Resolution Actions

Since conflicts are captured by the ac-alternatives within ac-tuples, community members can resolve them by selecting which ac-alternatives they believe in and which they do not. This can be done through the following two types of resolution actions:

- *Mark an ac-alternative as wrong.* This corresponds to removing (i.e., deleting) the ac-alternative. This action allows users to partially resolve a conflict even when they do not have the knowledge to fully resolve it. For instance, in our running example, if Clint was listed with 3 mailing addresses, in Carmel, Burbank and Paris, a user could restrict the conflicting values by removing Paris, even if she could not decide between Carmel and Burbank. On the other hand, when she can fully resolve a conflict, she can carry out the second type of resolution action:

- *Mark an ac-alternative as right.* This corresponds to marking all remaining ac-alternatives as wrong. As such it can be simulated by a set of ‘mark wrong’ actions. However to facilitate faster resolution, Ricolla offers it conveniently as a separate action. Note that this action specifies that an ac-alternative is right only w.r.t. the other ac-alternatives *currently* in the same ac-fragment. It will not remain right if additional ac-alternatives are added to the fragment. However users can still ask the system to consider an ac-alternative as *always* right (even under updates to the ac-fragment) by formulating an appropriate resolution policy (described in the next section).

Both actions can be carried out directly on the interface corresponding to Ricolla’s data model (by selecting an ac-alternative and choosing one of the two actions). This simplicity is not a coincidence but a result of the ac-database model.
As discussed in Section 4.4, one of the main requirements for our data model was to enable an interface that supports easy resolutions. Other proposed data models, although more expressive, do not satisfy this requirement, as conflicts are hidden within provenance formulas and the possible ways to resolve them are far from obvious.

An important feature of our resolution actions is that they have the expected behavior of reducing the set of conflicts. If we denote the action of marking an ac-alternative $a$ of an ac-database $I$ as right and wrong by $r^a_{\text{right}}(I)$ and $r^a_{\text{wrong}}(I)$, respectively, then the following corollary holds:

**Corollary 5.** Executing a ‘mark wrong’ resolution action on an ac-database $I$, decreases the set of possible worlds of $I$. If $a$ is an ac-alternative appearing within $I$ then the following holds:

$$PWorls(r^a_{\text{wrong}}(I)) \subset PWorls(I)$$

A similar result holds for ‘mark right’ resolution actions.

The corollary assumes a natural restriction: A resolution action on an ac-alternative $a$ is only defined if $a$ appears within an ac-fragment which contains at least one additional alternative (since otherwise $a$ is not part of a conflict).

Having presented the semantics of the resolution actions, let us now describe their implementation in the system. As explained in the introduction, community members should be allowed to disagree on how conflicts should be resolved. To facilitate this need, each individual’s resolution actions do not directly affect the community ac-database. Instead they update the ‘User Actions’ store, presented in Section 4.4.3. In particular, marking an ac-alternative $a$ as wrong, updates $a$’s ‘User Actions’ table with the information that the current user has deleted it. On the other hand, marking an ac-alternative as right is implemented by marking the remaining ac-alternatives within the same fragment as wrong.

Storing user actions in a separate storage allows each user to decide independently which data of the community ac-database she wants to see. To this end, users can write resolution policies that create their own view of the community
4.5.2 Resolution Policies

As we discussed in the architecture section (Section 4.3), a community member does not see the entire ac-community database. Instead she creates her own view of the database by writing a set of resolution policies. A resolution policy is a rule that summarizes a set of resolution actions based on high-level criteria, such as provenance or timestamps. Figure 4.9 shows the general form of such a policy. It consists of two parts: a) a condition which is checked against all ac-alternatives appearing in the ac-database to create a set of ac-alternatives of interest and b) a resolution action of one of the two types described above that is going to be applied to all the selected alternatives.

To enable flexible resolution policies, the policy’s condition can be expressed over both the data in the ac-database and the actions in the ‘User Actions’ tables. To enable formulating conditions over both data and actions, the Visual Policy Builder depicts their combined schema, shown on the left side of Figure 4.9. The combined schema is shown as an XML schema: Indentation levels denote parent-child relationships and ‘*’ symbols correspond to repeating elements. The schema follows the hierarchical definition of our data model: An ac-database consists of

<table>
<thead>
<tr>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
</tr>
<tr>
<td>acDatabase</td>
</tr>
<tr>
<td>acRelation*</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>acTuple*</td>
</tr>
<tr>
<td>acFragment*</td>
</tr>
<tr>
<td>acAlternative*</td>
</tr>
<tr>
<td>attribute*</td>
</tr>
<tr>
<td>name</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>action*</td>
</tr>
<tr>
<td>user</td>
</tr>
<tr>
<td>timestamp</td>
</tr>
<tr>
<td>action</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>right</td>
</tr>
<tr>
<td>wrong</td>
</tr>
</tbody>
</table>
ac-relations, ac-relations consist of ac-tuples etc. The most interesting part though appears below ac-alternatives. An ac-alternative element in this schema contains not only the alternative’s attributes with their values but also a set of user actions corresponding to that alternative’s ‘User Actions’ table. Please note that neither the ac-database nor the ‘User Actions’ table are stored as XML data conforming to this schema. This is a just a conceptual schema created solely for the purpose of specifying resolution policies.

**Conditions.** To specify a resolution policy, users place appropriate conditions on the attributes of the combined schema. The attributes that can be conditioned are shown in italics and the set of possible values that can be used in the condition appear on the right of each such attribute (separated with an OR (‘|’) character). These conditions restrict the set of ac-alternatives that are going to be marked by the policy as either right or wrong. The user can select the desired set of ac-alternatives in 3 ways: First, by placing a condition on the *ac-relation name*. In this case, she can select only those alternatives that appear in a specified relation (e.g. all ac-alternatives in ac-relation Actor). Second, by placing a condition on the *attribute name or value*. This allows her to select all alternatives whose schema contains (among others) an attribute of the specified name and/or value (e.g. all ac-alternatives with the value ‘Burbank’ for attribute City). Finally, by placing a condition on the *action’s attributes* (user, timestamp and action). This can be used to select alternatives manipulated by a certain user (e.g. all ac-alternatives inserted by Lara).

Of special interest are the ‘MIN’/’MAX’ values that can be used to condition an attribute value or a timestamp. Let us explain their semantics through an example. If we condition the attribute name with ‘Height’ and its value with ‘MIN’, then Ricolla will look for all ac-alternatives which contain the minimum value for attribute ‘Height’ among all other alternatives in the same fragment. This corresponds to Lara’s resolution policy described in the introduction, which selects the minimum height for each actor.
Default Policy. When a user first enters Ricolla she sees the community ac-database through a default resolution policy. This policy, depicted in Figure 4.10 for Lara, reflects her resolution actions, by marking as wrong all alternatives that she has marked as wrong. A user can at any time either discard this policy, or enter one or more new policies. All policies specified by the user will be executed in parallel to create the user’s view of the community ac-database.

Features. The resolution policy language is expressive enough to capture many common use cases. Moreover it satisfies the following requirements, derived from the needs of users in a collaborative conflict resolution system:

First of all, it allows both schema dependent and schema independent policies. Users sometimes want to resolve conflicts in the same way, regardless of the relation or the attribute they appear in. The default policy is such an example. In other cases, users want to treat conflicts in a specific part of the schema specially. Ricolla’s policy language allows users to easily specify policies for both scenarios.

Second, it enables collaboration between users, while allowing them to maintain different opinions. By specifying a condition on the user actions, a community member can easily avoid resolving certain conflicts and instead leverage the resolution actions carried out by her peers. If on the other hand, she disagrees with the other members, she can change the resolution policy to reflect that.

Semantics. Given an ac-database $I$ and a resolution policy $P$, executing $P$ on $I$ returns a new ac-database created by executing the resolution action specified in the policy on all ac-alternatives returned by the policy’s condition. This can
be straightforwardly extended to running a set of resolution policies on an ac-
database. Note that the order in which resolution actions within a policy or across
policies are executed on the ac-database does not affect the result, since both ‘mark
as wrong’ and ‘mark as true’ alternatives are translated to a set of ac-alternative
deletions.

The semantics of a resolution policy are implemented in terms of queries on
the underlying representation of an ac-database. In particular, given a relational
representation as described in Section 4.8, resolution policies are translated into
appropriate SQL statements as described in Section 4.8.4.

4.6 Query Answering

One of the main goals of Ricolla is to allow users to get the answers to
their queries even when the system contains conflicting data. Substantial amount
of existing work, referred to as consistent query answering (see Section 4.9 for a
thorough discussion of related work) suggests to return only the answers that are
consistent w.r.t. a set of constraints expressed over the database schema. However
this is against the focus of our system, where a user should be allowed to see all
data (including the conflicting/inconsistent ones) and be allowed to resolve the
conflicts according to her beliefs.

This is why in our system we chose to present to the user not some chosen
single query result but the set of all possible query results (i.e. the set of the
answers to the query against each of the possible worlds of the ac-database). In the
following we formally define these query answering semantics, which we call possible
answers to the query. All definitions employ set-semantics for query answering.

Definition 7. Possible Answers: Given a query $Q$ and an ac-database $I$, the
possible answers to $Q$ over $I$ is the following set of possible worlds:

$$PAnswers_Q(I) = \{Q(DB) | DB \in PWorl ds(I)\}$$

A nice property of the possible answers is that they naturally extend the
standard flat relational answering semantics: Consider the ac-database represent-
tation of a flat relational database, where each flat tuple is wrapped inside an
ac-tuple (i.e. for each flat tuple we create an ac-tuple with a single ac-fragment which contains the flat tuple as its unique ac-alternative). Then running the query over the ac-database representation of the flat database under the possible answer semantics and subsequently flattening it yields the same result as running the query over the flat database under the relational answering semantics:

**Corollary 6.** Let \( \text{flat2ac} \) be a function that takes as input a flat relational database and returns an ac-database in which each flat tuple is treated as a single alternative within a separate ac-tuple and \( \text{ac2flat} \) its inverse. Then for each flat database \( I \) and a query \( Q \) over the schema of \( I \), the following holds:
\[
\text{ac2flat}(PAnswers_Q(\text{flat2ac}(I))) = Q(I)
\]

Note that the possible answers to a query are defined as a set of possible worlds. This means that in general the answer to the query might not be representable in our data model. We identify a sufficiently large class of queries and ac-databases for which the possible answers to the query of an ac-database can be represented in our data model. This uniformity, referred to as closure of the data model under that query language, guarantees that for the particular class of queries we can use the same data model to represent both the base ac-relations and the query results. Moreover query results can easily be stored back to an ac-database if we want to allow this functionality.

We will show that our model is closed under a subclass of conjunctive queries that join only on attributes that cannot have multiple possible values within any ac-tuple. To this end we start by defining the join attributes of a schema:

**Definition 8.** **JOIN ATTRIBUTES:** Given an ac-database schema \( S \), the set of join attributes is the set of attributes which are allowed to participate in query joins.

Note that we are not interested in which exact pairs of attributes are going to be joined together but rather whether an attribute can appear in a join on not. Given a set of join attributes over a schema, an ac-database over the same schema is join-consistent if it does not have uncertainty in the value of the join attributes. Formally:
Definition 9. **Join-Consistent Ac-Database:** An ac-database $I$ over schema $S$ is said to be Join-Consistent with respect to a set of join attributes over the same schema, if it satisfies the following properties:

- For every ac-tuple in $I$ all join attributes of the corresponding ac-relation appear within the same fragment of the ac-tuple and
- all alternatives of that fragment are non-empty and identical when projected on the join attributes.

For instance, in our running example we expect the users of the system to formulate queries joining on movie titles and actor names. Therefore we consider as the set of join attributes the set $J = \{\text{Movie.Title}, \text{Actor.Name}, \text{MovieActor.Title} \text{ and MovieActor.Name}\}$. The ac-database containing the 3 relations shown in Figures 4.2, 4.4 and 4.5 is join-consistent w.r.t. $J$ as it does not contain multiple values for any of the join attributes. For example, the ac-tuples in relation Movie in Figure 4.4 contain only a single possible value for the movie title (which is a join attribute).

Let us now define the query language under which the data model is closed. We are interested in join-consistent $CQ_1^=$ queries. Such a query is a subset of conjunctive queries, defined below that contains joins only on the join attributes. In the following we define both classes of join-consistent queries and $CQ_1^=$ queries separately:

**Definition 10. Join-Consistent Query:** A query $Q$ expressed over an ac-database schema $S$ is called Join-Consistent w.r.t. a set of join attributes over the same schema, if $Q$ contains joins only on the join attributes.

**Definition 11. $CQ_1^=$:** The class $CQ_1^=$ contains all select-join relational algebra queries containing at most one instance of each relation. Selections involve equalities between attributes and constants.

Having defined the set of ac-databases and queries of interest, we can now formulate the closure result:
Theorem 13. Closure: Let $J$ be a set of join attributes, $Q$ a join-consistent (w.r.t. $J$) $CQ^w_1$ query and $I$ a join-consistent (w.r.t. $J$) ac-database. Then the possible answers of $Q$ over $I$ can be represented by an ac-relation.

In Ricolla we exploit this closure property to make sure that the query results can be represented in the same data model and interface as the base data. To be able to rely on the closure property, Ricolla offers mechanisms that guarantee that the ac-database is join-consistent w.r.t. a predefined set of join attributes and that the queries posed by the user fall in the class $CQ^w_1$. In particular, Ricolla allows the community initiator to specify the set $J$ of join attributes at schema design time. Subsequently, it guarantees that the ac-database remains join-consistent w.r.t. $J$ by rejecting ac-DML statements that would lead to a violation of the join-consistency. For instance, a user trying to insert a second ac-alternative for the title of the movie “Dirty Harry” in Figure 4.4 will receive a warning that this action is not possible. Moreover Ricolla ensures that users formulate only join-consistent (w.r.t. $J$) $CQ^w_1$ queries by offering an appropriate visual query builder.

The visually constructed queries, guaranteed to satisfy the closure property, are then passed to the Backend, which computes their possible answers as an ac-relation. A detailed description of the algorithm used to compute the possible answers is given in Section 4.8.3.

4.7 Resolution Actions on Query Results

In Section 4.5.1 we saw how users can resolve conflicts by carrying out conflict resolution actions on the base data (i.e. on ac-tuples that appear in the community database). However a significant number of users access the community data indirectly through queries. These users would ideally want to resolve the conflicts on the query answer instead of the base data. To facilitate them, the system allows resolution actions to be carried out on the query answer. Each such action is then automatically translated to a set of resolution actions on the base data that have the same effect. This is related to the classical view update problem [BS81], where updates on views (i.e. queries) over a database are translated to
direct updates over the database.

The following definition captures the properties that a set of resolution actions over the base data has to satisfy in order to be considered a translation of a resolution action over the query answer.

**Definition 12.** **Translation:** Let $I$ be an ac-database and $Q$ a $CQ^I_1$ query over the schema of $I$. A translation of a resolution action $r_Q$ over $PAnswers_Q(I)$ is a set of ‘mark as wrong’ resolution actions $R_I$ over $I$, s.t.:

- $r_Q(PAnswers_Q(I)) = PAnswers_Q(R_I(I))$ and
- $\not\exists R'_I \text{ s.t. } R'_I \subset R_I$ and $r_Q(PAnswers_Q(I)) = PAnswers_Q(R'_I(I))$

The first clause captures the requirement for the translation to have the same effect on the data as the resolution action over the query answer. In particular it requires the translation to be such that, carrying it out on the base data and subsequently running the query yields the same outcome as carrying out the original resolution action on the query answer. The second clause requires the translation to be the minimal set of resolution actions over the base data that satisfy the first property. This ensures that a translation resolves only those conflicts that are required for it to have the same effect as the original resolution action.

It turns out that such a translation always exists and is unique:

**Theorem 14.** **Uniqueness of the Translation:** Let $I$ be an ac-database, $Q$ a $CQ^I_1$ query over the schema of $I$ and $r_Q$ a resolution action over $PAnswers_Q(I)$. The translation of $r_Q$ always exist and it is unique.

### 4.8 Implementation

In this section we describe the algorithms powering the system. In a first iteration we implemented the system on top of a RDBMS. The benefits from this

---

2For the purpose of the definition, we extend the definition of the execution of a single resolution action to sets of resolution actions. Note that if this set consists of ‘mark as wrong’ resolutions only, the order of executing actions in the set is irrelevant, as any order yields the same result.
approach are twofold: First, it allows leveraging the query answering capabilities and optimizations offered by RDBMSs. Second, it allows enterprises hosting Ri-colla to reuse their existing RDBMS infrastructure. As part of our future work, we plan to investigate techniques of natively implementing the system.

The system consists of 4 main components: a) storing an ac-database as a flat database, called flattening, b) retrieving an ac-database from its flat representation, called nesting, c) answering queries and d) executing resolution policies.

4.8.1 Flattening Ac-Relations

To store an ac-database in a RDBMS, the system should convert each ac-relation to one or more flat relations. This process should be invertible to ensure that an ac-relation can be reconstructed from its flat representation.

First cut. A straightforward approach of storing an ac-relation in a RDBMS is by flattening it in the nested relational sense: Let us consider each ac-tuple as a nested relation consisting of a single tuple whose inner tables correspond to its ac-fragments. Then we can convert this ac-tuple to a set of flat tuples simply by applying the unnest operator, as defined in the nested relational algebra [RKS88]. The result consists of all flat tuples that are represented by that ac-tuple (i.e. the ac-tuple’s interpretations as defined in Section 4.4.2). Thus an easy way to keep an ac-relation in a RDBMS is by creating a flat relation that stores the interpretations of all its ac-tuples.

For example, using this approach the ac-tuple about Clint Eastwood shown in Figure 4.2 can be stored in a flat relation through the four rows shown in Figure 4.11. The four rows are the ac-tuple’s interpretations (shown in Figure 4.7) augmented with additional information (such as tuple and fragment identifiers) that ensure that the flattening operation can be reversed.

However this flattening algorithm has a major drawback: Through unnesting, it essentially creates the cartesian product between all nested tables within an ac-tuple. This introduces a lot of redundancy in the flat relation, leading to space requirements that can be exponentially larger than the ac-relation it represents. In particular, for each ac-tuple containing $f$ ac-fragments with $n$ ac-alternatives
Figure 4.11: Flat representation of the ac-tuple in Figure 4.2 through the ac-tuple’s interpretations

<table>
<thead>
<tr>
<th>tid</th>
<th>Name</th>
<th>Nfid</th>
<th>Height</th>
<th>Hfid</th>
<th>City</th>
<th>Cfid</th>
<th>Zip</th>
<th>Zfid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clint</td>
<td>1</td>
<td>1.85</td>
<td>2</td>
<td>Burbank</td>
<td>3</td>
<td>91...</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Clint</td>
<td>1</td>
<td>1.85</td>
<td>2</td>
<td>Carmel</td>
<td>3</td>
<td>93...</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Clint</td>
<td>1</td>
<td>1.88</td>
<td>2</td>
<td>Burbank</td>
<td>3</td>
<td>91...</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Clint</td>
<td>1</td>
<td>1.88</td>
<td>2</td>
<td>Carmel</td>
<td>3</td>
<td>93...</td>
<td>3</td>
</tr>
</tbody>
</table>

(a) Actor

Ricolla’s flattening algorithm. To overcome this problem we introduce an alternative flattening algorithm that does not incur the exponential space overhead. The main idea behind it is to store each ac-alternative only once (together with additional data to make the process invertible).

To accomplish that, an ac-relation is decomposed into a set of flat relations. If all ac-tuples within an ac-relation were partitioned into fragments in the same way, then it would suffice to create as many flat relations as the number of fragments per ac-tuple. Each flat table would hold the ac-alternatives of all fragments sharing the same schema. However, an ac-relation allows ac-tuples to contain fragments of different schemas. Therefore the flattening algorithm employs a separate flat relation for each attribute. For instance, Figure 4.12 shows the result of flattening the Clint Eastwood ac-tuple of Figure 4.2.

As can be seen through the example, each ac-relation with \( n \) attributes is flattened into \( n + 1 \) relations: 1 tuple relation and \( n + 1 \) attribute relations. The *tuple relation* holds a single flat tuple for each ac-tuple in the ac-relation. This tuple contains a unique tuple identifier (\( tid \)) (for internal use by the system) as well as the identifiers of the fragments (denoted by \( fid_i \)) in which each of the ac-tuple’s
attributes appear. The values of attributes within ac-tuples on the other hand are stored in the respective attribute relation. Each tuple in an attribute relation holds a value of that attribute together with the identifiers of the fragment (\textit{fid}) and the alternative within that fragment (\textit{aid}) in which it appears. The resulting algorithm \textit{Flatten}, is shown below:

\begin{algorithm}
  \caption{Flatten}
  \begin{algorithmic}[1]
    \Require ac-relation $R_{ac}$ with \( n \) attributes $\text{attr}_1, \ldots, \text{attr}_n$
    \Ensure $n + 1$ flat relations $R_{\text{tuple}}$, $R_1$, $\ldots$, $R_n$
    \Begin
      1. Create a flat relation $R_{\text{tuple}}$ with $n + 1$ attributes:
        \begin{itemize}
          \item $\text{tid}$, $\text{fid}_1$, $\ldots$, $\text{fid}_n$
        \end{itemize}
      2. Create $n$ flat relations $R_1$, $\ldots$, $R_n$, each with 3 attributes:
        \begin{itemize}
          \item $\text{fid}$, $\text{aid}$, $\text{value}$
        \end{itemize}
      3. \For each ac-tuple $t$ in $R_{ac}$
      4. \hspace{1em} let $\text{tid} = \text{fresh\_tid}(R_{ac})$
      5. \hspace{1em} let $\text{fids} = $ new array of length $n$
      6. \hspace{1em} \For each ac-fragment $f$ in $t$
      7. \hspace{2em} let $\text{fid} = \text{fresh\_fid}(R_{ac}, f)$
      8. \hspace{2em} \For each ac-alternative $a$ in $f$
      9. \hspace{3em} let $\text{aid} = \text{fresh\_aid}(f)$
     10. \hspace{3em} \For each attribute $\text{attr}_i$ in $f$'s schema
     11. \hspace{4em} let $\text{fids}[i] = \text{fid}$
     12. \hspace{3em} Create a new flat tuple $t_{\text{flat}}$ in $R_i$ with:
     13. \hspace{4em} $t_{\text{flat}}.\text{fid} = \text{fid}$
     14. \hspace{4em} $t_{\text{flat}}.\text{aid} = \text{aid}$
     15. \hspace{4em} if $a$ is an empty ac-alternative \textbf{then}
     16. \hspace{5em} $t_{\text{flat}}.\text{value} = \bot$
     17. \hspace{4em} else
   \End
  \End
\end{algorithmic}
\end{algorithm}

Figure 4.12: Flat representation of the ac-tuple in Figure 4.2 using the algorithm \textit{Flatten}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
$\text{fid}$ & $\text{aid}$ & $\text{value}$ \\
\hline
1 & 1 & Clint \\
\hline
2 & 1 & 1.85 \\
2 & 2 & 1.88 \\
\hline
3 & 1 & Burbank \\
3 & 2 & Carmel \\
\hline
3 & 1 & 91522 \\
3 & 2 & 93921 \\
\hline
\end{tabular}
\caption{R_1: Name}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
$\text{fid}$ & $\text{aid}$ & $\text{value}$ \\
\hline
3 & 1 & 91522 \\
3 & 2 & 93921 \\
\hline
\end{tabular}
\caption{R_3: City}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
$\text{fid}$ & $\text{aid}$ & $\text{value}$ \\
\hline
2 & 1 & 1.85 \\
2 & 2 & 1.88 \\
\hline
\end{tabular}
\caption{R_2: Height}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
$\text{fid}$ & $\text{aid}$ & $\text{value}$ \\
\hline
3 & 1 & 91522 \\
3 & 2 & 93921 \\
\hline
\end{tabular}
\caption{R_4: ZipCode}
\end{table}
18. \[ t_{\text{flat}. \text{value}} = a.\text{attr}_i \]
19. Create a new flat tuple \( t_{\text{flat}} \) in \( R_{\text{tuple}} \) with:
20. \[ t_{\text{flat}. \text{tid}} = \text{tid} \]
21. \[ t_{\text{flat}. \text{fid}_i} = \text{fids}[i], \forall i = 1, ..., \text{n} \]
end

The procedures \( \text{fresh} \_\text{tid}(R) \), \( \text{fresh} \_\text{fid}(R, f) \) and \( \text{fresh} \_\text{aid}(f) \) generate tid, fid and aid values, respectively. In particular, procedures \( \text{fresh} \_\text{tid}(R) \) and \( \text{fresh} \_\text{aid}(f) \) return a value that is guaranteed to be unique across all invocations of the respective procedure with the same argument. In this way, the algorithm generates tid values that are unique within each ac-relation and aid values that are unique within each ac-fragment. The procedure \( \text{fresh} \_\text{fid}(R, f) \) is an exception: It returns a value that is guaranteed to be unique across all its invocations with the same parameter \( R \), unless its argument \( f \) has the same color as an existing fragment \( f' \). In that case the procedure returns the value previously returned for \( f' \). By creating the same \( \text{fid} \) for fragments that share the same color, the flattening algorithm preserves the coloring information. Finally, the algorithm accounts for empty ac-alternatives as follows: Each empty alternative is treated as a non-empty alternative whose attributes contain a special value \( \perp \). This allows both empty and non-empty alternatives to be treated uniformly through the procedure described above. To make sure that \( \perp \) models only the empty value, we consider it to be a value outside the domain of possible values of ac-tuples.

The advantage of this approach is that its space requirements are polynomial to the size of the original ac-relation:

**Corollary 7.** Let \( n \) be the number of values (including duplicates) appearing in the visual representation of an ac-relation \( R_{\text{ac}} \). The flattened representation of \( R_{\text{ac}} \) returned by the algorithm Flatten contains \( O(n) \) values.

### 4.8.2 Nesting Flat Relations

Given the flat representation of an ac-relation, it is straightforward to reconstruct it. The \( \text{Nest} \) procedure (omitted due to simplicity) simply traverses the relation table and follows the fragment id pointers to the individual attribute tables to construct the ac-fragments and ac-alternatives.
It turns out that the result of algorithm Flatten contains enough information for an ac-relation to be reconstructed:

**Theorem 15.** For any ac-relation $I$ flattening and subsequently nesting it yields the original ac-relation:

$$I = Nest(Flatten(I))$$

Note that flattening preserves not only the set of possible worlds represented by the ac-relation but also its exact structure. In particular, even though there can be several ac-relations corresponding to the same set of possible worlds, these will have different flattened representations.

### 4.8.3 Answering Queries

Query answering can be generally accomplished in two different ways: The query answering algorithm can operate either directly on the ac-database or on its flattened representation. However running the algorithm on the ac-database involves first recreating the entire ac-database from its flat representation. This can be a huge overhead, especially if the ac-database is large and the query is very selective. To overcome this problem, we choose the second approach: The query $Q_{ac}$ over the ac-database is rewritten to a set of queries that can be executed inside the RDBMS over the flattened representation of the ac-database. In this way, we avoid nesting the entire database and in parallel we leverage the query answering and optimization capabilities of the RDBMS. Subsequently the flat query results (which correspond to the flattened representation of the possible answers to $Q_{ac}$) can be passed as input to the nesting algorithm to construct the possible answers to $Q_{ac}$. Figure 4.13 shows both the semantics of query answering and the way it is implemented in the system. In the following we describe the rewriting procedure that translates a query over an ac-database to queries over its flattened representation.

Our system employs an algorithm for the translation of all queries in the class $CQ_1^\tau$. In this section we describe the algorithm only for queries that do not involve joins but the algorithm can be straightforwardly extended to also take joins
into account. Although a restricted setting, it still allows us to demonstrate some of the challenges that arise in the query rewriting process. The query rewriting algorithm, which we describe next, is shown below:  

\begin{algorithm}
\textbf{algo} : \texttt{TranslateQuery}
\begin{algorithmic}[1]
\State \textbf{input:} query $Q_{ac} \in CQ^{+}$ with a single relation $R_{ac}$ containing $n$ attributes
\State \textbf{output:} $n+1$ queries $Q_{tuple}, Q_{1}, ..., Q_{n}$
\Function{begin}
\State 1. let $C$ be the set of selection conditions in $Q_{ac}$
\State 2. $Q_{\text{removedAlt}} = \bigcup_{c \in C} \pi_{fid, aid} \sigma_{\neg c}(rel(c))$, where $rel(c)$ the flat relation created by algorithm Flatten for the attribute selected in condition $c$
\State 3. let $attr_{1}, ..., attr_{n}$ be the attributes in $R_{ac}$’s schema
\State 4. for $i = 1, ..., n$
\State 5. $Q_{\text{remain}}^{i} = R_{i} \bowtie Q_{\text{removedAlt}}$
\State 6. $Q_{\text{emptyAlt}}^{i} = \pi_{fid, \text{max}+1 \text{ as } aid, \bot \text{ as value}}(\pi_{fid}(Q_{\text{remain}}^{i}) \bowtie \pi_{fid}(Q_{\text{removedAlt}}))$
\State 7. $Q_{i} = Q_{\text{remain}}^{i} \cup Q_{\text{emptyAlt}}^{i}$
\State 8. $Q_{tuple} = R_{tuple}$,
\State 9. for $i = 1, ..., n$
\State 10. $Q_{tuple} = Q_{tuple} \bowtie \pi_{fid, i \text{ as } fid}(Q_{i})$
\Function{end}
\end{algorithmic}
\end{algorithm}

Since the flattened representation of the possible answers of $Q_{ac}$ consists of $n + 1$ relations (where $n$ the number of attributes in the head of $Q_{ac}$), the

3The algorithm uses the generalized projection operator, which supports outputting the result of functions on attributes rather than only attributes, as described in [SKS]
original query $Q_{ac}$ over an ac-database schema $S$ needs to be translated to $n + 1$ queries $Q_{tuple}$, $Q_1$, ..., $Q_n$ over the flattened representation of $S$. $Q_{tuple}$ returns the tuple relation, as defined in Section 4.8.1 and each of the $Q_i$’s returns an attribute relation.

The main challenge in query rewriting is that a query selection does not affect only a single attribute relation but in general a set of them. Indeed, by putting a selection $c$ on an attribute $a$ of an ac-relation, we are essentially removing entire ac-alternatives that do not satisfy the selection condition. These ac-alternatives might contain values not only for attribute $a$ but also for other attributes and are thus stored in multiple attribute relations. Therefore the query rewriting algorithm should translate a single selection over an attribute to selections over all attribute relations.

To this end, the algorithm first identifies all ac-alternatives that have to be removed due to either of the selections (Step 2). This is accomplished as follows: For each selection condition involving attribute $a$, it creates a query that returns all alternatives in the attribute table of $a$ that do not satisfy the condition. By taking the union of all these queries, it creates a query $Q_{removedAlt}$ that computes all ac-alternatives that need to be removed due to failing either of the selection conditions. Step 5 exploits the query generated in Step 2 to generate a query that computes all values that remain in each of the attribute tables after applying the selections. The new query $Q_{remain}^i$ employs an antisemijoin to return all tuples in the attribute table for attribute $attr_i$ that do not correspond to ac-alternatives in $Q_{removedAlt}$. However, this does not suffice as selections can introduce empty alternatives. In particular, when a selection leads to the removal of some (but not all) ac-alternatives within an ac-fragment, the query result should contain an empty ac-alternative within that fragment. This empty alternative depicts that the corresponding tuple might not exist as it might not satisfy the selection condition. Steps 6 and 7 add such empty ac-alternatives to the attribute tables created through the queries $Q_{remain}^i$, $i = 1, ..., n$. The resulting queries $Q_i$ return the attribute tables in the flat representation of the query result.

Finally, steps 8-10 create the query $Q_{tuple}$ returning the tuple table. The
tuple table should contain only those ac-tuples whose all ac-fragments still exist in the attributes table. To this end, the query $Q_{\text{tuple}}$ computes the result of semijoining the original tuple table with each of the $Q_i$’s.

Given the algorithm $\text{TranslateQuery}$, the system can translate a query $Q_{\text{ac}}$ over an ac-database schema $S$ to a set of queries over the flattened representation of $S$. Please note that, similar to other query rewriting algorithms, the algorithm $\text{TranslateQuery}$ does not evaluate the queries. Instead it simply computes their definitions. Thus the translation algorithm needs to be invoked only once when a new query is designed over the ac-database schema and not every time the query is executed. At query evaluation time, the precomputed queries returned by the query rewriting algorithm can be evaluated to return the flattened representation of the possible answers to the original query.

### 4.8.4 Executing Resolution Policies

Ricolla supports resolution policies by translating them to appropriate queries over the flattened representation of the community ac-database. Executing these queries yields the flattened representation of the result of applying the resolution policies over the ac-database. The generated queries (omitted due to lack of space) are similar in function to queries created by the $\text{TranslateQuery}$ algorithm, described above: Operating on the attribute tables, they find all ac-alternatives that satisfy the conditions set by the resolution policy, and then either they remove them from the attribute tables (if the policy specifies a ‘mark as wrong’ action) or they remove all other ac-alternatives (for ‘mark as right’ actions). Moreover, similarly to queries generated by $\text{TranslateQuery}$, they update the relation tables to reflect the changes to the attribute tables. The main difference between the relational queries generated for user queries and those for resolution policies, is that the latter can also operate on the UserActions relation, which stores the contents of the ‘User Actions’ tables of all alternatives.

Once Ricolla translates a policy down to relational queries, it can generate the User View. However the latter is not materialized as such. Instead, whenever a user asks a query, her query (translated to queries over the store) is composed
with the queries corresponding to the resolution policy. By not materializing the User View, Ricolla can on-the-fly create only the part of the view that is of interest to the queries.

### 4.9 Related Work

Researchers have looked at several aspects of the problem of managing conflicting data:

**Querying inconsistent data.** A significant amount of work [ABC99, FFM05] (nicely summarized in [Cho07]) focuses on querying a relational database that contains inconsistent data. To this end, they propose the *Consistent Query Answering* semantics. According to them, a query returns the set of tuples that appear on the answers to the query against *all* minimal repairs of the original database. A minimal repair of a database is a way to convert it to a consistent database with the least amount of changes. Intuitively the consistent query answers are those tuples that are guaranteed to exist, regardless of how the inconsistency is resolved.

Ricolla on the other hand, provides more informative answers, containing also data that are inconsistent. This allows the users to inspect conflicts and resolve them. Note that the consistent query answers can be easily inferred from the possible answers through a linear scan: They correspond to ac-tuples with a single alternative in each fragment.

Recently, these frameworks were extended to take only preferred repairs into account [SCM06]. Preferences are specified through rules resembling Ricolla’s resolution policies.

**Modeling inconsistent data.** Lately there have been a lot of proposals on data models to represent uncertain or probabilistic data. However, as explained in Section 4.4.4, these data models are not directly applicable to our setting.

**End-to-end systems for managing inconsistent data.** Finally, several systems were proposed as attempts to solve the problem of inconsistent data management. However they cannot be used effectively in the Ricolla setting. OR-
CHESTRA [TI06] allows users to reconcile data while allowing disagreement. However, in contrast to Ricolla, it applies to P2P scenarios, where community users have their own local databases. Moreover, it takes inconsistencies into account and tries to resolve them only when a user decides to reconcile the data published by their peers. After each reconciliation period, local databases are consistent (and conflicts that could not be resolved discarded). On the other hand, Ricolla’s goal is to always keep and display to the user all inconsistencies, until she can resolve them. Other systems, such as HumMer [NBBW06] and Fusionplex [MA06] allow inconsistency resolution in the context of data fusion. They provide resolution policy languages but they lack a formally defined model for summarizing inconsistent data. Moreover they are designed with a single user in mind and are thus not applicable in collaborative scenarios.

4.10 Future Work

In this section, we present two possible future extensions of Ricolla. The first extends the query language supported by the tool. The second allows Ricolla to support bulk inserts and to be integrated into a community-oriented integration system to work seamlessly with RIDE, as outlined in Section 1. We discuss next each of these directions in detail.

4.10.1 Approximation of Query Results

As we explained in Section 4.6, the intuitiveness of Ricolla’s data model has a negative effect on the class of queries supported by the tool. Due to the limited expressiveness of the ac-database, this model is closed only under a very restricted class of queries. More expressive queries are not supported as their answers may correspond to sets of possible worlds not expressible as ac-databases. A possible remedy to this problem would be to replace the ac-database with a more expressive data model or at the extreme, a data model that is complete and can express all finite sets of possible worlds. However, for reasons explained in Section 4.4.4, this would lead to a data model not intuitive enough and therefore not suitable for the
Instead we envision a different solution which keeps the data model intact by modifying instead the query answering semantics. Whenever users run a query whose result cannot be represented as an ac-database, the system will return an ac-database that is an optimal approximation of the set of possible worlds that form the query answer. An ac-database $I$ is considered to be an optimal approximation of a set of possible worlds $W$ iff $PWorlds(I)$ is a minimal superset of $W$. By choosing a superset of the query answer, we guarantee that the user will get to see at least all possible worlds present in the query answer. In other words, the approximation will show all possible databases that can be created by resolving all inconsistencies in the data and possible a few that cannot be reached, since it will miss some correlations that exist in the query answer. Moreover, by computing the minimal such superset, we guarantee that the approximation stays as close as possible to the actual query result, given the expressivity of the ac-database.

We plan to investigate algorithms for computing optimal approximations of sets of possible worlds as part of our future work.

### 4.10.2 Support for GLAV-based Information Integration

As we discussed in Section 1, in its current incarnation Ricolla only supports insertion of data manually one piece at a time. On the other hand, RIDE supports the integration of multiple sources through GLAV constraints. To realize our vision of community-oriented information systems, Ricolla has to be extended to support the integration setting employed by RIDE. This extension requires work on two components:

**Algorithm for converting a flat community database to an ac-database.** In the online database setting described in this chapter, users directly augment the community ac-database by either creating new ac-tuples or adding ac-alternatives to existing tuples. However in the community-oriented integration setting described in Chapter 2, this is not the case: Sources contribute relational data to a flat relational global database. To allow users to resolve inconsistencies in
such a setting, we have to develop appropriate algorithms that convert the global
database into an ac-database by grouping together into ac-tuples all flat tuples
that correspond to alternatives of the same object.

If the global database contains primary key constraints and the sources
provide globally agreed values for the primary key attributes, then this procedure
can be as simple as creating an ac-tuple for each set of flat tuples that agree on
the primary key value: This ac-tuple will consist of a single ac-fragment containing
as ac-alternatives all flat tuples with the corresponding key value. If the global
database does not contain any primary key attributes or the sources provide dirty
values for those attributes, we can still use the same technique by also incorporating
entity resolution algorithms that find tuples corresponding to the same entity, as
those described in [BGMM+09].

However the proposed algorithm has an important drawback: By creating
ac-tuples with a single ac-fragment, it misses the opportunities for compact rep-
resentation of alternatives offered by the ac-database. Indeed, in some scenarios
it is natural to split an ac-tuple to multiple ac-fragments. For instance, if sources
always provide only the height of an actor or pairs of city and zip code (but not
both simultaneously), we can safely assume that the possible height values are
orthogonal to the address values and should therefore be stored in separate ac-
fragments. Designing an algorithm that makes such inferences in the general case
is a non-trivial task that we plan to tackle as part of our future work.

**Extension of the ac-database and query answering to support incompleteness.** RIDE’s setting creates an additional complication when we try
to incorporate Ricolla into it. Due to the expressiveness of the GLAV mapping
constraints, the global database is not a flat relational database, as we assumed
above. Instead it contains additional information in the form of incompleteness:
A source may not provide values for some of the attributes (in which case these
attributes could take any possible value) or a source may even express equalities
between the values of two attributes without providing those values (recall our
running example in Section 2.5, in which B&N specified that it sells its books at
the suggested retail price, without providing the suggested retail price).

To support GLAV mappings, the ac-database should therefore be extended to also support these types of incompleteness. Moreover the query answering semantics and algorithms should be modified accordingly to handle queries on incomplete data such as joins on B&N’s regular price for books and their suggested retail price. We plan to investigate this issue as part of our future work.

4.11 Conclusions

Online communities need an online database system that allows them to collaboratively resolve inconsistencies in an ‘as-you-go’ fashion, while tolerating inconsistencies. In this chapter, we present the design of Ricolla; an end-to-end system for this task. While being formally grounded, Ricolla enables a natural workflow for the processing of conflicting data: Users can model and inspect conflicts (through the ac-database model), query the data (based on the possible answer semantics) and resolve data either collaboratively or individually in two different granularities (through resolution actions and policies).

This chapter contains material prepared for publication by Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou and Vasilis Vassalos. Parts of it appear in a demo paper that is currently under review for ICDE 2010.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

Recent years saw the proliferation of collaborative applications that aim in allowing communities to share their data through a single point of access. Flickr, Youtube and Facebook became hugely successful for multimedia and unstructured data. On the other hand, in the structured data front, the first applications appeared only very recently and are still in their infancy. Current commercial offerings of online databases either support very limited forms of structured data (such as a single table or a list) or are straightforward ports of traditional database management systems to the web, failing to recognize the different needs of communities compared to enterprises.

This dissertation attempts to change this picture by proposing a paradigm that provides a natural workflow for communities to integrate their data into a global database. The main idea behind the novel paradigm is to start from the widely studied EII systems and distribute the integration tasks that were carried out by the central integration authorities to the individual community members. While doing so, we account for the lack of sophistication of the individual peers as well as their lack of overview of the system by providing tools that assist them in the execution of each task. We describe the decentralization of the two main integration tasks; the source registration and the inconsistency resolution.

The source registration is pushed to the source owners. We recognize that
the motivating force behind the registration of a source is the desire to make its data available to application queries running over the global schema and present RIDE. RIDE is an interactive registration tool that allows a source owner to negotiate the trade-off between the visibility of her source to application queries and the cost of cleaning her data to make them conform to the global schema. We model this trade-off as different self-reliance levels and present decision procedures for each of them. Moreover we design and implement RIDE as a GLAV schema mapping tool augmented with a suggestion component that explains to the user how to extend the current registration of her source to reach the desired contribution level. Finally, we implement a prototype of the tool and perform an experimental evaluation to prove that the tool forms a viable solution even for communities with a large number of registered sources. Moreover, for completeness, we present theoretical results on alternative definitions of the self-reliance levels.

The second task that is decentralized is the inconsistency resolution, which is pushed to the individual clients of the integration system. To this end, we present Ricolla; an online database system for inconsistency resolution. We formally define a novel data model suitable for the user interface of an inconsistency resolution tool, which captures inconsistencies as a set of possible worlds and presents them in a summarized form. We define the query answering semantics in this system and prove that the data model is closed under a sufficiently large class of queries, allowing thus query answers to be represented in the same model. We also present a set of resolution actions and policies, which enable users to resolve inconsistencies at different granularities. Finally, we present algorithms to implement this system on top of a relational database management system.

5.2 Future Work

Given the current trend of online databases and collaboration systems for communities, the area of community-oriented information integration has great potential for impactful future work.

A few possible extensions of our work are listed in Sections 2.9 and 4.10 for
RIDE and Ricolla, respectively.

On the source registration side, we can extend RIDE to allow users to receive suggestions for contribution to an entire application (consisting of a set of queries), instead of a single application query (see Section 2.9.2). Moreover, to support the lifetime of a community, which typically evolves over time, we can employ schema mapping composition techniques to support evolution of a community’s global schema, as described in Section 2.9.1.

On the inconsistency resolution side, we can extend Ricolla to handle more expressive queries by computing approximations of query results, as discussed in Section 4.10.1. Additionally, we can extend our inconsistency resolution tool to work not only in an online database setting (where data are manually inserted by hand) but also in the GLAV-based integration environment supported by RIDE. By making this extension (outlined in Section 4.10.2), we will enable a system where RIDE and Ricolla coexist, thus realizing our vision of a community-oriented integration system.

Once an end-to-end CII system is implemented and deployed in real communities, an entirely new set of opportunities for future work will open up. By inspecting user data, we will be able to extract patterns that exist in communities and design additional functionality that exploits these patterns to provide an even more guided assistance to users both for source registration and for inconsistency resolution. For instance, in real life sources are not expected to provide a random subset of attributes of the global schema. Instead they will most probably fall into a relatively small set of clusters, each containing sources with similar data. For example, in an integration system about books, most of the sources are expected to be bookstores, newspapers/magazines and publishers. By clustering the already registered sources into groups, RIDE could predict the type of a new source. Consequently, instead of offering generic suggestions on how to increase contribution (some of which may not be applicable to the particular type of source), it could provide more informed and guided suggestions. Similar principles apply also in the inconsistency resolution task. In scientific communities, where members have differing opinions, we expect most members to fall into sets of ‘schools’ that follow
a particular opinion. By computing these clusters, Ricolla could predict the inconsistency resolution actions of a user (based on the cluster in which she belongs), thus relieving her from the need to carry out the inconsistency resolution herself.

We plan to investigate these possible directions and look at new opportunities to help users integrate their data by relying on real data to discover the relationships that exist between sources or between the opinions of different clients.
Appendix A

Proofs

Proof. (of Theorem 2) The proof is a reduction from the Post Correspondence Problem (PCP). Let $L_1 = \{u_i\}_{1 \leq i \leq n}$, $L_2 = \{v_i\}_{1 \leq i \leq n}$ be lists of words over an alphabet $\Sigma$ (i.e. $u_i \in \Sigma^*$, $v_i \in \Sigma^*$, $1 \leq i \leq n$). A solution to PCP is a sequence of indexes $i_1, ..., i_m$ s.t. $u_{i_1}u_{i_2}...u_{i_m} = v_{i_1}v_{i_2}...v_{i_m}$. The string $u_{i_1}u_{i_2}...u_{i_m}$ is called the expansion of this solution. In order to prove the theorem, for any PCP instance we create an integration system (i.e. a source schema $S$, a target schema $G$, a set of assertions $A$, a set of source constraints $\Delta_S$, a set of target constraints $\Delta_G$ and a set of mapping constraints $M$) s.t. a PCP instance has a solution iff the corresponding integration system is potentially inconsistent.

For ease of exposition we first create an auxiliary integration system $IS_{aux}$ shown in Figures A.1 and A.2. Then we extend it to the actual integration system $IS$ used in the reduction as shown in Figure A.3. Note that the constructed integration systems (both $IS$ and $IS_{aux}$) contain just a single registered source and therefore they contain a single source schema $S$ and a single registration $M$.

Let us first present $IS_{aux}$. Source relation $E_S$ is the edge relation of a labeled directed graph with $E_S(s, l, t)$ describing the edge from node $s$ to $t$ with label $l$. The intention is to represent a word $w = a_1...a_p$ (where $a_i$ are letters) by a chain of the form $E_S(x_1, a_1, x_2), ..., E_S(x_p, a_p, x_{p+1})$. On the target schema, target relation $C$ is intended to contain tuples $C(s_u, s_v, i, t_u, t_v)$ if from pair of nodes $s_u, s_v$ we can reach nodes $t_u, t_v$ following paths representing $u_i, v_i$, respectively. Additionally, target relation $R$ should contain a tuple $R(t_u, t_v)$ if nodes $t_u, t_v$ are reachable
• Source schema \( S = \{ E_S: \text{3-ary} \} \)

• Target schema \( G = \{ C: \text{5-ary}, R: \text{2-ary} \} \)

• Set of assertions \( A = \emptyset \)

• Set of source constraints \( \Delta_S = \{(U^i \subseteq V^i)|1 \leq i \leq 2\} \)

\[
U^1_\delta(s, l_1, t_1, l_2, t_2) : - E_S(s, l_1, t_1), E_S(s, l_2, t_2) \\
V^1_\delta(s, t, l, t) : - E_S(s, t) \\
U^2_\delta(t, s_1, l_1, s_2, l_2) : - E_S(s_1, l_1, t), E_S(s_2, l_2, t) \\
V^2_\delta(t, s, l, s) : - E_S(s, l)
\]

• Set of target constraints \( \Delta_G = \{(U^3 \subseteq V^3)\} \)

\[
U^3_\delta(x', y') : - R(x, y), C(x, y, i, x', y') \\
V^3_\delta(x', y') : - R(x', y')
\]

Figure A.1: Proof of Theorem 2: Auxiliary Integration System \( IS_{aux} \): Schemas, Assertions, Source Constraints and Target Constraints

from the same node \( s \) by paths representing \( u_{i_1}...u_{i_k} \) and \( v_{i_1}...v_{i_k} \), respectively for some indexes \( i_1, ..., i_k \). Therefore a tuple of the form \( R(x, x) \) means that node \( x \) is reachable by some node \( s \) both by a path representing \( u_{i_1}...u_{i_k} \) and one representing \( v_{i_1}...v_{i_k} \). Since however the graph contains only chains, these paths will coincide and represent the expansion of a solution to the PCP. Thus there exists a tuple of the form \( R(x, x) \) iff PCP has a solution.

Let us now move to the integration system \( IS \). It is an extension of \( IS_{aux} \) such that there exists a target instance in \( IS \) that satisfies all mapping and target constraints iff \( IS_{aux} \) does not contain a tuple of the form \( R(x, x) \). Therefore \( IS \) is potentially inconsistent iff \( IS_{aux} \) contains a tuple of the form \( R(x, x) \). However, since the latter happens exactly when PCP has a solution, \( IS \) is potentially inconsistent iff PCP has a solution.

The above semantics are specified as follows: The source constraints \( (U^i_\delta \subseteq V^i_\delta), 1 \leq i \leq 2 \) restrict the source instances to graphs consisting of a set of disjoint chains and cycles. Note that the source instances cannot be restricted to graphs
Set of mappings \( M = \{(U^i_G \subseteq V^i_G)| 1 \leq i \leq 2n\}\)

foreach \(1 \leq i \leq n\), let \(u_i = a_1...a_k\) and \(v_i = b_1...b_l\)

\[U^i_G(x_1, y_1, i, x_{k+1}, y_{l+1}) : -\]

\[E_S(x_1, a_1, x_2), E_S(x_2, a_2, x_3), ..., E_S(x_k, a_k, x_{k+1}),\]

\[E_S(y_1, b_1, y_2), E_S(y_2, b_2, y_3), ..., E_S(y_l, b_l, y_{l+1})\]

\[V^i_G(x_1, y_1, i, x_2, y_2) : - C(x_1, y_1, i, x_2, y_2)\]

let \(l_0\) be a letter not in \(\Sigma\)

foreach \(1 \leq i \leq n\), let \(u_i = a_1...a_k\) and \(v_i = b_1...b_l\)

if one of \(u_i, v_i\) is a prefix of the other, then

\[U^{i+n}_G(x_k, y_l) : -\]

\[E_S(s_0, l_0, s), \]

\[E_S(s, a_1, x_1), E_S(x_1, a_2, x_2), ..., E_S(x_{k-1}, a_k, x_k),\]

\[E_S(s, b_1, y_1), E_S(y_1, b_2, y_2), ..., E_S(y_{l-1}, b_l, y_l)\]

\[V^{i+n}_G(x_1, y_1) : - R(x_1, y_1)\]

else remove the constraint (\(U^{i+n}_G \subseteq V^{i+n}_G\))

Figure A.2: Proof of Theorem 2: Auxiliary Integration System \(IS_{aux}\): Mapping Constraints

containing only chains, since chains and cycles are indistinguishable by first-order formulas. Furthermore, the mapping constraints (\(U^i_G \subseteq V^i_G\), \(1 \leq i \leq n\)) are used to capture the intended meaning for relation \(C\).

Target constraint (\(U^3_G \subseteq V^3_G\)) and mapping constraints (\(U^i_G \subseteq V^i_G\), \(n + 1 \leq i \leq 2n\)) implement the semantics of relation \(R\), which, according to its definition, should contain the transitive closure of \(C\). The recursive step to obtain the transitive closure is described by (\(U^3_G \subseteq V^3_G\)) and the base case of the recursion is captured by constraints (\(U^i_G \subseteq V^i_G\), \(n + 1 \leq i \leq 2n\)). The base case consists of pairs of nodes \(t_u, t_v\) s.t. they are reachable from a node \(s\) (with an incoming edge label \(l_0 \notin \Sigma\)) both through a path representing \(u_i\) and through one representing \(v_i\), where \(u_i\) is a prefix of \(v_i\) or vice versa. The prefix requirement is due to the fact that if \(i_1, ..., i_m\) is a solution to the PCP, then one of \(u_{i_1}, v_{i_1}\) will be a prefix of the other. Moreover the requirement that the start node \(s\) has an incoming edge
Same as $IS_{aux}$ shown in Figures A.1 and A.2 with an additional target relation $I$ and two additional target constraints involving $I$:

- **Source schema** $S = \{E_S: 3$-ary $\}$
- **Target schema** $G = \{C: 5$-ary, $R: 2$-ary, $I: 2$-ary $\}$
- **Set of assertions** $A = \emptyset$
- **Set of source constraints** $\Delta_S = \{(U_i^S \subseteq V_i^S)|1 \leq i \leq 2\}$
- **Set of target constraints** $\Delta_G = \{(U_i^G \subseteq V_i^G)|3 \leq i \leq 5\}$
  
  $U_4^S(x) : \neg R(x,x)$
  $V_4^S(x) : \neg I(x,1), I(x,2)$
  $U_5^S(x,y_1,y_2) : \neg I(x,y_1), I(x,y_2)$
  $V_5^S(x,y,y) : \neg I(x,y)$

- **Set of mappings** $M = \{(U_i^G \subseteq V_i^G)|1 \leq i \leq 2n\}$

Figure A.3: Proof of Theorem 2: Integration System $IS$ used in the Reduction

labeled $l_0 \notin \Sigma$ avoids considering a path from $s$ to $t$ as the expansion of a solution to the PCP when there is a path from $s$ to $t$ through $u_{i_1}...u_{i_m}$ and one through $v_{i_1}...v_{i_m}$ but one of these paths goes around a cycle on which $s$, $t$ are located more times than the other.

Finally, target constraints $(U_i^G \subseteq V_i^G), 4 \leq i \leq 5$ establish the connection between $IS$ and $IS_{aux}$ by specifying that target relation $I$ contains a key constraint which is violated whenever there exists a tuple of the form $R(x,x)$.

Proof. (of Lemma 1) By definition, $Suf_Q(R)$ holds if and only if the certain answer to $Q$ is non-empty on some database $DB$, if and only if $rew_R(Q)$ returns a non-empty answer on $DB$, if and only if $rew_R(Q)$ is satisfiable.

Proof. (of Theorem 5) By Theorem 1, it suffices to check satisfiability of $rew_R(Q)$. Since by Theorem 12, we have that $rew_R(Q) \in UCQ^\omega$, this amounts to finding at least one satisfiable individual conjunct $C$ in the union. $C$ is satisfiable if and
only if there is no equality between distinct constants which follows by transitivity, symmetry and reflexivity from the equality atoms of $C$.

Proof. (of Theorem 6) The proof is a reduction from the Post Correspondence Problem (PCP). Let $L_1 = \{u_i\}_{1 \leq i \leq n}$, $L_2 = \{v_i\}_{1 \leq i \leq n}$ be lists of words over an alphabet $\Sigma$ (i.e. $u_i \in \Sigma^*$, $v_i \in \Sigma^*$, $1 \leq i \leq n$). A solution to PCP is a sequence of indexes $i_1, ..., i_m$ s.t. $u_{i_1}u_{i_2}...u_{i_m} = v_{i_1}v_{i_2}...v_{i_m}$. The string $u_{i_1}u_{i_2}...u_{i_m}$ is called the expansion of this solution. In order to prove the theorem, for any PCP instance we create an instance of the Self Sufficiency Problem (SSP) (i.e. a global schema $G$, a set of target constraints $\Delta_G$, a source registration $R = (S, M)$ and a query $Q$) s.t. PCP has a solution iff $Suf_Q(R)$ holds.

For ease of exposition we first present a reduction using one target constraint that is neither a PK nor a RIC and then change the SSP instance to include only PKs and RICs as target constraints.

- **Local schema** $S = \{E_S: 3$-ary $\}$
- **Global schema** $G = \{E^1_G: 3$-ary, $E^2_G: 3$-ary, $C: 5$-ary, $R: 2$-ary $\}$
- **Set of target constraints** $\Delta_G = \{(U^1_{\delta} \subseteq V^1_{\delta}), (U^2_{\delta} \subseteq V^2_{\delta}), \delta_3\}$

\[
\begin{align*}
U^1_{\delta}(s, l_1, t_1, l_2, t_2) &::= E^1_G(s, l_1, t_1), E^1_G(s, l_2, t_2) \\
V^1_{\delta}(s, l, t, l, t) &::= E^1_G(s, l, t) \\
U^2_{\delta}(t, s_1, l_1, s_2, l_2) &::= E^2_G(s_1, l_1, t), E^2_G(s_2, l_2, t) \\
V^2_{\delta}(t, s, l, s, l) &::= E^2_G(s, l, t) \\
\delta_3 &= (\forall x, y, i[R(x, y) \land C(x, y, i, x', y') \rightarrow R(x', y')])
\end{align*}
\]

Figure A.4: Proof of Theorem 6: Self Sufficiency Problem Instance for First Cut: Schemas and Target Constraints

**First Cut.** The constructed SSP instance that contains a non-PK and non-RIC target constraint $\delta_3$ is shown in Figures A.4 and A.5 (the first figure shows the schemas and target constraints of the instance and the second the mapping constraints and the application query). Source relation $E_S$ is the edge relation of a labeled directed graph with $E_S(s, l, t)$ describing the edge from node $s$ to
• **Set of mappings** \( M = \{(U^i_G \subseteq V^i_G)|1 \leq i \leq 2n + 2\} \)

\[
\begin{align*}
U^i_G(s, l, t) & : = E(S(s, l, t) \text{ and } V^i_G(s, l, t) : = E^i_G(s, l, t) \\
V^i_G(s, l, t) & : = E(S(s, l, t) \text{ and } V^i_G(s, l, t) : = E^i_G(s, l, t)
\end{align*}
\]

foreach \( 1 \leq i \leq n \), let \( u_i = a_1...a_k \) and \( v_i = b_1...b_l \)

\[
\begin{align*}
U^{i+n+2}_G(x_k, y_l) & : = E(S(s_0, l_0, s), \ E(S(s, a_1, x_1), E(S(x_1, a_2), x_2), \ldots, E(S(x_k, a_k, x_k)), \ E(S(s, b_1, y_1), E(S(y_1, b_2, y_2), \ldots, E(S(y_{l-1}, b_l, y_l)) \ V^{i+n+2}_G(x_1, y_1) & : = R(x_1, y_1)
\end{align*}
\]

else remove the constraint \( U^{i+n+2}_G \subseteq V^{i+n+2}_G \)

• **Application Query** \( Q() : = R(x, x) \)

Figure A.5: Proof of Theorem 6: Self Sufficiency Problem Instance for First Cut: Mappings and Application Query

\( t \) with label \( l \). The intention is to represent a word \( w = a_1...a_p \) (where \( a_i \) are letters) by a chain of the form \( E_S(x_1, a_1, x_2), \ldots, E_S(x_p, a_p, x_{p+1}) \). On the global schema, target relation \( C \) is intended to contain tuples \( C(s_u, s_v, i, t_u, t_v) \) if from pair of nodes \( s_u, s_v \) we can reach nodes \( t_u, t_v \) following paths representing \( u_i, v_i \), respectively. Additionally, target relation \( R \) should contain a tuple \( R(t_u, t_v) \) if nodes \( t_u, t_v \) are reachable from the same node \( s \) by paths representing \( u_{i_1}...u_{i_k} \) and \( v_{i_1}...v_{i_k} \), respectively for some indexes \( i_1, \ldots, i_k \). Finally, query \( Q \) asks for the existence of a tuple \( R(x, x) \), i.e. a node \( x \) reachable by some node \( s \) both by a path representing \( u_{i_1}...u_{i_k} \) and one representing \( v_{i_1}...v_{i_k} \). Since however the graph contains only chains, these paths will coincide and represent the expansion
of a solution to the PCP. Thus $Q$ has nonempty certain answers for some source instance (i.e. $\text{Suf}_Q(R)$ holds) iff PCP has a solution.

The above semantics are specified as follows: The mapping constraints $(U_i^G \subseteq V_i^G), 1 \leq i \leq 2$ and target constraints $(U_i^\delta \subseteq V_i^\delta), 1 \leq i \leq 2$ restrict the source instances to graphs consisting of a set of disjoint chains and cycles. Note that the source instances cannot be restricted to graphs containing only chains, since chains and cycles are indistinguishable by first order formulas. Furthermore, the mapping constraints $(U_i^G \subseteq V_i^G), 3 \leq i \leq n+2$ are used to capture the intended meaning for relation $C$.

Finally, target constraint $\delta_3$ and mapping constraints $(U_i^G \subseteq V_i^G), n+3 \leq i \leq 2n+2$ implement the semantics of relation $R$, which, according to its definition, should contain the transitive closure of $C$. The recursive step to obtain the transitive closure is described by $\delta_3$ and the base case of the recursion is captured by constraints $(U_i^G \subseteq V_i^G), n+3 \leq i \leq 2n+2$. The base case consists of pairs of nodes $t_u, t_v$ s.t. they are reachable from a node $s$ (with an incoming edge label $l_0 \not\in \Sigma$) both through a path representing $u_i$ and through one representing $v_i$, where $u_i$ is a prefix of $v_i$ or vice versa. The prefix requirement is due to the fact that if $i_1, \ldots, i_m$ is a solution to the PCP, then one of $u_{i_1}, v_{i_1}$ will be a prefix of the other. Moreover the requirement that the start node $s$ has an incoming edge labeled $l_0 \not\in \Sigma$ avoids considering a path from $s$ to $t$ as the expansion of a solution to the PCP when there is a path from $s$ to $t$ through $u_{i_1} \ldots u_{i_m}$ and one through $v_{i_1} \ldots v_{i_m}$ but one of these paths goes around a cycle on which $s, t$ are located more times than the other.

Let us now modify the SSP instance, so that $\Delta_G$ consists of PKs and RICs only.

**Second Cut.** The new instance is shown in Figures A.6 and A.7. While using the same source schema, we are now capturing the semantics of both the relations $C$ and $R$ used in the first cut by a single target relation $C'$. The intention is that if from a node $s_0$ (which has an incoming $l_0$ edge), we can reach nodes $s_u, s_v$ through chains representing $u_{i_1} \ldots u_{i_k}$ and $v_{i_1} \ldots v_{i_k}$, respectively for some indexes $i_1, \ldots, i_k$ and additionally from nodes $s_u, s_v$ we can reach $t_u, t_v$ through chains
Figure A.6: Proof of Theorem 6: Self Sufficiency Problem Instance for Second Cut: Shemas and Target Constraints

- Local schema $S = \{E_S: 3$-ary $\}$
- Global schema $G = \{E_G^1: 3$-ary, $E_G^2: 3$-ary, $C': 7$-ary, $C'': 3$-ary, $C''': 1$-ary $\}$
- Set of target constraints $\Delta_G = \{(U^i_\delta \subseteq V^i_\delta)|1 \leq i \leq 6\}$

| $U^3_\delta$ $(s, l, t_1, l_2, t_2)$ | $E^3_G(s, l, t_1), E^3_G(s, l_2, t_2)$ |
| $V^3_\delta$ $(s, l, t_1, l, t)$ | $E^3_G(s, l, t)$ |
| $U^2_\delta$ $(t, s_1, l_1, s_2, l_2)$ | $E^2_G(s_1, l_1, t), E^2_G(s_2, l_2, t)$ |
| $V^2_\delta$ $(t, s_1, l_1, s_2, l)$ | $E^2_G(s, l, t)$ |
| $U^3_\delta$ $(s_0, t_u, t_v)$ | $C''(s_0, s_u, s_v, i, t_0, t_u, t_v)$ |
| $V^3_\delta$ $(t_0, t_u, t_v)$ | $C''(s_0, s_u, s_v, i, t_0, t_u, t_v)$ |
| $U^4_\delta$ $(s_0, s_u, s_v)$ | $C'(s_0, s_u, s_v, i, t_0, t_u, t_v)$ |
| $V^4_\delta$ $(o, t_1, t_2)$ | $C'''(o, t_1, t_2)$ |
| $U^5_\delta$ $(t_0, t_u, t_v)$ | $C''(s_0, s_u, s_v, i, t_0, t_u, t_v)$ |
| $V^5_\delta$ $(o, t_1, t_2)$ | $C''(o, t_1, t_2)$ |
| $U^6_\delta$ $(t_1, t_2, o_1, o_2)$ | $C'''(o_1, t_1, t_2), C'''(o_2, t_1, t_2)$ |
| $V^6_\delta$ $(t_1, t_2, o, o)$ | $C''(o, t_1, t_2)$ |

Figure A.6: Proof of Theorem 6: Self Sufficiency Problem Instance for Second Cut: Shemas and Target Constraints

describing $u_i$, $v_i$, respectively for some index $i$, then there should exist a tuple $C'(s_0, s_u, s_v, i, s_0, t_u, t_v)$. Relation $C'''$ is a unary relation containing the nodes that can serve as starts of a chain (i.e. nodes with an incoming edge labeled with $l_0$).

Thus the query $Q$ is now asking for the existence of a node $t$, which is reachable from some node $t_0$ that can serve as the start of a chain both via a path labeled $u_{i_1}...u_{i_k}$ and one labeled $v_{i_1}...v_{i_k}$. For the reasons outlined in the first cut, $\text{Su}_{\delta} Q(R)$ holds iff PCP has a solution.

The intended semantics for the relations are again implemented through constraints. As in the first cut, mapping constraints $(U^i_G \subseteq V^i_G), 1 \leq i \leq 2$ and target constraints $(U^i_\delta \subseteq V^i_\delta), 1 \leq i \leq 2$ disallow the existence of anything different from disjoint chains and cycles in the graph described by $E_S$. 
• Set of mappings $M = \{(U^i_G \subseteq V^i_G)|1 \leq i \leq 2n + 3\}$

\[
U^1_G(s, l, t) : - E_S(s, l, t) \text{ and } V^1_G(s, l, t) : - E^l_G(s, l, t)
\]

\[
U^2_G(s, l, t) : - E_S(s, l, t) \text{ and } V^2_G(s, l, t) : - E^2_G(s, l, t)
\]

foreach $1 \leq i \leq n$, let $u_i = a_1...a_k$ and $v_i = b_1...b_l$

\[
U^{i+2}_G(x_1, y_1, i, x_{k+1}, y_{l+1}) : -
\]

\[
E_S(x_1, a_1, x_2), E_S(x_2, a_2, x_3), ..., E_S(x_k, a_k, x_{k+1}),
\]

\[
E_S(y_1, b_1, y_2), E_S(y_2, b_2, y_3), ..., E_S(y_l, b_l, y_{l+1})
\]

\[
V^{i+2}_G(x_1, y_1, i, x_{2}, y_2) : - C'(z_1, x_1, v_1, z_2, x_2, y_2)
\]

let $l_0$ be a letter not in $\Sigma$

foreach $1 \leq i \leq n$, let $u_i = a_1...a_k$ and $v_i = b_1...b_l$

if one of $u_i, v_i$ is a prefix of the other, then

\[
U^{i+n+2}_G(s, x_k, y_l) : -
\]

\[
E_S(s_0, l_0, s),
\]

\[
E_S(s, a_1, x_1), E_S(x_1, a_2, x_2), ..., E_S(x_{k-1}, a_k, x_k),
\]

\[
E_S(s, b_1, y_1), E_S(y_1, b_2, y_2), ..., E_S(y_{l-1}, b_l, y_l)
\]

\[
V^{i+n+2}_G(s, x_1, y_1) : - C'(s, x_1, y_1, z_1, z_2, z_3, z_4)
\]

else remove the constraint ($U^{i+n+2}_G \subseteq V^{i+n+2}_G$)

\[
U^{2n+3}_G(s_0, l_0, x) : - E_S(s_0, l_0, x)
\]

\[
V^{2n+3}_G(x) : - C'''(x)
\]

where $l_0$ the same letter used above

• Application Query $Q() : - C'(s_0, s_u, s_v, i, t_0, t, t), C'''(t_0)$

Figure A.7: Proof of Theorem 6: Self Sufficiency Problem Instance for Second Cut: Mappings and Application Query

Additionally, mapping constraints ($U^i_G \subseteq V^i_G$), $3 \leq i \leq n + 2$ map into $C'$ what was mapped into $C$ in the first cut. Similarly constraints ($U^i_G \subseteq V^i_G$), $n + 3 \leq i \leq 2n + 2$ bring into $C'$ what was mapped into $R$ in the first cut to form the base case for the recursion. The semantics of the recursive step are the following: If there exists a tuple $C''(s_0, s_u, s_v, i, t_0, t, t)$ then $t_0$ should be equal to $s_0$ and moreover, whenever an additional tuple $C'(s'_0, t_u, t_v, i', t'_u, t'_v)$ appears in $C'$, $s'_0$ should be equal to $t_0$. These semantics are ensured through constraints
$(U^i_i \subseteq V^i_i)$, $3 \leq i \leq 6$. Finally the intended semantics of $C''''$ are captured by mapping constraint $(U^{2n+3}_g \subseteq V^{2n+3}_g)$.

Proof. (of Theorem 7) By definition, registration $R_{n+1}$ is now complementary iff it is not self-sufficient and there are source extents $\overline{DB}, DB_{n+1}$ s.t. $cert_{\overline{DB},DB_{n+1}}(Q) \setminus cert_{\overline{DB}}(Q) \neq \emptyset$.

(i) The proof of Theorem 5 shows that $R_{n+1}$ is self-sufficient iff $rewr_{R_{n+1}}(Q)$ is satisfiable.

(ii) There are source extents $\overline{DB}, DB_{n+1}$ s.t. $cert_{\overline{DB},DB_{n+1}}(Q) \setminus cert_{\overline{DB}}(Q) \neq \emptyset$ iff there are $\overline{DB}, DB_{n+1}$ with

$rewr_{\overline{DB},DB_{n+1}}(Q) \setminus rew_{\overline{DB}}(Q) \neq \emptyset$

iff

there are $\overline{DB}, DB_{n+1}$ with

$rewr_{\overline{DB},DB_{n+1}}(Q) \setminus rew_{\overline{DB}}(Q) \neq \emptyset$ (since $rew_{\overline{DB}}(Q)$ does not mention the schema of $DB_{n+1}$)

iff

$rewr_{\overline{DB},DB_{n+1}}(Q)$ is not contained in $rew_{\overline{DB}}(Q)$. \hfill \Box

Proof. (of Theorem 8) Given global schema $G$, local schema $S$, registration $R = (S, M)$ and query $Q$ formulated against $G$, we construct the new local schema $G'$, two local schemas $S_1, S_2$ with registrations $R_1, R_2$, and query $Q'$ against $G'$ such that $R$ is self-sufficient w.r.t. $Q$ if and only if $R_2$ is now-complementary w.r.t. $R_1$ and $Q'$: $Suf_R(Q)$ iff $NComp^Q_{R_1}(R_2)$.

We obtain $G'$ by adding to $G$ a fresh, zero-ary relation $N$. $Q'$ is obtained from $Q$ by adding the atom $N()$ to the body. $S_1$ consists of a single, zero-ary relation $O$, and the registration mapping $M_1$ is given by $M_1 = \{(U \subseteq V)\}$ with $U() : - O()$ and $V() : - N()$. $S_2$ and $R_2$ coincide with $S$ and $R$, respectively.

Notice that given the same extent to the sources of schema $S$, query $Q'$ returns the same answer as $Q$ if the extent of $O$ is not empty, and returns the empty answer otherwise.
We observe that \( \text{Suf}_{Q'}(R_2) \) does not hold, as \( M_2 \) does not map into \( N \) and therefore regardless of the extent of source 2, there is always a target database in which \( N \) is empty, and one in which it is not (contains the empty tuple). The certain answers to \( Q' \) are therefore always empty if source 2 is alone in the system. That is, \( R_2 \) is not self-sufficient.

Similar reasoning shows that source 1 alone contributes no certain answers to \( Q' \) regardless of its extent, as it does not map into the other relations appearing in \( Q' \).

Therefore, in order for source 2 to contribute additional certain answers to \( Q' \) in the presence of source 1, it suffices to contribute any one tuple. Any such answer would be an answer to query \( Q \) as well. We conclude that \( \text{Suf}_{R}(Q) \) iff \( \text{NComp}^D_{R_1}(R_2) \).

\[ \text{Proof of Theorem 9} \] We use the concept of canonical target instances found in the literature [FKMP03, FKP03]. We give its definition for \( \Delta_G = \emptyset \).

**Canonical Target Instances.** Given source registrations \( \overline{R} \) and corresponding extents \( \overline{DB} \), a canonical target instance of \( \overline{R} \) and \( \overline{DB} \) is a minimal global instance that satisfies the source mappings and contains specially constructed Skolems for all attributes whose values are not restricted by \( \overline{R} \).

Note that due to different naming of the Skolems, many canonical target instances may exist. We denote by \( \text{can}(\overline{R}; \overline{DB}) \) any one of them.

Such an instance can be easily created by treating the mapping constraints as rules that, starting from tuples of the sources, generate tuples in \( G \). For a detailed explanation of the construction of a canonical target instance, see [FKMP03, FKP03, YP04].

This work also shows that by running \( Q \) on any canonical target instance and removing from the result all tuples containing Skolems (denoted by \( Q(\text{can}(\overline{M}; \overline{DB}))\downarrow \)) we can compute the set of certain answers to \( Q \):

**Lemma 2.** For \( \Delta_G = \emptyset \) and \( Q \in UCQ^= \) the following holds: \( \text{cert}_{\overline{DB}}^M(Q) = Q(\text{can}(\overline{R}; \overline{DB}))\downarrow \)

\[ \text{Proof. (of Theorem 9)} \] (if) This direction immediately follows by the definition of Later Complementary.
(only if) Assume that $LComp^Q_R(R_{n+1})$ holds. Then by definition there exist registrations $\overline{R}$ and source extents $\overline{DB}$ and $DB_{n+1}$ s.t. $\text{cert}^R_{\overline{DB},DB_{n+1}}(Q) \setminus \text{cert}^R_{\overline{DB}}(Q) \neq \emptyset$. We will, based on the existence of $\overline{R}$, $\overline{DB}$, show that there also exists an extent $DB_{id}$ of a source with the identity registration such that

\[
\text{cert}^{M_{id},M_{n+1}}_{DB_{id},DB_{n+1}}(Q) \setminus \text{cert}^{M_{id}}_{DB_{id}}(Q) \neq \emptyset.
\]

We obtain $DB_{id}$ by building $\text{can}(\overline{R}'; DB')$ and replacing any distinct Skolem value in it by a distinct fresh constant (i.e. a constant that appears neither in $\overline{DB}$, $DB_{n+1}$, nor in $Q$). Since $R_{id}$ copies the source extent to the target without leaving any value unspecified, it is easy to see that $\text{can}(R_{id}; DB_{id}) = DB_{id}$. Thus $\text{cert}^{M_{id}}_{DB_{id}}(Q) = Q(\text{can}(R_{id}; DB_{id})) \downarrow = Q(DB_{id})$. However by construction, $DB_{id}$ contains all information provided by $\text{can}(\overline{R}; DB)$ plus specific new values instead of Skolems. Thus $Q(DB_{id})$ (which has been shown equal to $\text{cert}^{M_{id}}_{DB_{id}}(Q)$) contains all tuples of $Q(\text{can}(\overline{R}; DB))\downarrow$ plus some new ones, each of which contains at least one of the fresh constants introduced in $DB_{id}$. Thus if we denote by $A$ this set of new tuples, the following holds:

\[
\text{cert}^{M_{id}}_{DB_{id}}(Q) \setminus Q(\text{can}(\overline{R}; DB))\downarrow = A.
\]

Using Lemma 2, this transforms to:

\[
\text{cert}^{M_{id}}_{DB_{id}}(Q) \setminus \text{cert}^R_{\overline{DB}}(Q) = A
\]

and combined with the fact that

\[
\text{cert}^R_{\overline{DB},DB_{n+1}}(Q) \setminus \text{cert}^R_{\overline{DB}}(Q) \neq \emptyset
\]

yields

\[
\text{cert}^R_{\overline{DB},DB_{n+1}}(Q) \setminus \text{cert}^{M_{id}}_{DB_{id}}(Q) \neq \emptyset
\]  \hspace{1cm} (A.1)

(since all tuples in $A$ contain fresh constants and thus they can not appear in $\text{cert}^R_{\overline{DB},DB_{n+1}}(Q)$).

We will now compare $\text{cert}^R_{\overline{DB},DB_{n+1}}(Q)$ and $\text{cert}^{M_{id},M_{n+1}}_{DB_{id},DB_{n+1}}(Q)$, which by Lemma 2 equal to $Q(\text{can}(\overline{R}, R_{n+1}; DB, DB_{n+1})) \downarrow$ and $Q(\text{can}(\overline{R}_{id}, R_{n+1}; DB_{id}, DB_{n+1})) \downarrow$ respectively.
\(DB_{n+1}\), respectively. As previously, by the construction of \(DB_{id}\) from \(\overline{DB}\) it follows that

\[
\text{cert}_{DB_{id}, DB_{n+1}}^{M_{id}, M_{n+1}}(Q) \supseteq \text{cert}_{DB_{id}, DB_{n+1}}^{M_{id}, M_{n+1}}(Q).
\]

Equations A.1 and A.2 yield:

\[
\text{cert}_{DB_{id}, DB_{n+1}}^{M_{id}, M_{n+1}}(Q) \setminus \text{cert}_{DB_{id}}^{M_{id}}(Q) \neq \emptyset.
\]

Hence \(NComp^Q_{\text{Rid}}(R_{n+1})\) holds.

\(\square\)

Proof. (of Theorem 10) Consider the registration \(R\) and a future registration \(R_f\). In the presence of both registrations, \(Q\) must have at least one certain answer, therefore it must have a match against the canonical instance as defined in the proof of Theorem 9. The image of \(Q\)'s atoms consists of tuples in the canonical instance corresponding to \(R, R_f\) (\(\text{can}_{R, R_f}\)). These tuples couldn’t all have existed in the canonical instance of \(R\) alone, \(\text{can}_R\), (since \(Q\) would have been self-sufficient), or in that of \(R_f, \text{can}_{R_f}\), (since \(R\) wouldn’t be now complementary with \(R_f\)). There must therefore exist a nonempty set of tuples in \(\text{can}_R\) such that some \(k\) attributes in them do not have the value needed to serve as image of \(Q\). These attributes are turned into the desired values once the registration \(R_f\) is added. The only way these values can be changed by \(R_f\) is if they are Skolems in \(\text{can}_R\). It is easy to see that, since each registration introduces its own Skolems, the only way \(R_f\) can affect Skolems introduced by \(R\) is by interference through PKs. That is, \(R_f\) can change some Skolem \(Sk\) into some value \(v\) only if \(\text{can}_R\) contains some tuple \(t\) in some relation \(P\) whose attribute set \(X\) is a key, such that none of \(t[X]\) are Skolems, and such that for some attribute \(A \notin X, t[A] = Sk\). \(R_f\) must insert into \(\text{can}_{R, R_f}\) a new \(P\)-tuple \(t'\) such that \(t[X] = t'[X]\) and \(t'[A] = v\). Note that \(P\) does not have to be mentioned by \(Q\), since Skolems can occur in several places in \(\text{can}_R\). \(Sk\) may occur in a tuple which will serve as image for \(Q\) once \(R_f\) arrives, and also in some (potentially distinct) \(P\)-tuple.

The value \(v\) may come from (i) the source \(f\), or (ii) it may be moved from another tuple in \(\text{can}_R\).

Case (i) requires source \(f\) to contribute data via some mapping of form \(D[Y] \subseteq P[X, A]\), where \(D[Y]\) is the query projecting \(D\) on its attribute set \(Y\) and
\(|Y| = |X| + 1\). Case (ii) requires some mapping of form \(D[X, Y] \subseteq \sigma_{A=B}(N[Y, B] \times P[X, A])\), where relation \(N\) is not necessarily distinct from \(P\), \(Y\) is the key for \(N\), and \(B\) is some non-key attribute of \(N\). The mapping “loads” the value \(v = t''[B]\) from some \(N\)-tuple \(t'' \in can_R\) by providing an \(N\)-tuple \(t'''\) which coincides with \(t''\) on the key value and which holds a Skolem in the \(B\) attribute. Due to the PK on \(N\), \(t''[B] = t'''[B] = v\). \(v\) is then “stored” into \(t[A]\), because the mapping provides a \(P\)-tuple \(t'\) agreeing with \(t\) on the key, and holding \(v\) in its \(A\) attribute.

The set of all \(k\) mappings as above can be summarized into a single mapping stating the inclusion of the Cartesian product of the left-hand queries into the Cartesian product of the right-hand queries. The size of \(D\) and of the mapping queries follows.
Bibliography


