Interactive Source Registration in Community-oriented Information Integration

Yannis Katsis
CSE Department
UC San Diego
ikatsis@cs.ucsd.edu

Alin Deutsch
CSE Department
UC San Diego
deutsch@cs.ucsd.edu

Yannis Papakonstantinou
CSE Department
UC San Diego
yannis@cs.ucsd.edu

ABSTRACT

Modern Internet communities need to integrate and query structured information. Employing current information integration infrastructure, data integration is still a very costly effort, since source registration is performed by a central authority which becomes a bottleneck. We propose the community-based integration paradigm which pushes the source registration task to the independent community members. This 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 her data contradict (or could it contradict in the future) the data of other sources? We introduce RIDE, a visual registration tool that extends schema mapping interfaces like that of MS BizTalk Server and IBM's Clio with a suggestion component that guides the source owner in the autonomous registration, assisting her in answering these questions. RIDE's implementation features efficient procedures for deciding various levels of self-reliance of a GLAV-style source registration for contributing answers to an application query and checking potential and definite inconsistency across sources.

1. INTRODUCTION

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 that 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 [3] and BIRN [1], aim to provide integrated access to the data of multiple laboratories and scientific communities. Such portals typically rely on traditional integration technology, and come at an often prohibitive setup and maintenance cost to the central integration authority. Indeed, the construction 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.

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

The other extreme consists of communities that publish unstructured data (text or multimedia files tagged with attribute-value pairs) with no integration functionality: published 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 flickr (http://www.flickr.com/), del.icio.us (http://del.icio.us/) and YouTube (http://www.youtube.com/). A salient feature contributing to their massive success is the low setup and maintenance cost due to the decentralized nature that allows new members to join autonomously without assistance from a central authority.

The above 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 traditional integration 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 cleanup of pertinent data. Graduate students and other researchers end up manually performing these tasks at a great productivity cost. This cost barrier is generally faced by many private, commercial, academic and even governmental communities.

We address this need by introducing the *community-based integration* paradigm which enables systems that integrate and query structured data into a virtual global database, at no cost to any central authority. This is achieved by decentralizing the setup and maintenance tasks, pushing them to the independent community members. In particular, in this paper we focus on assisting individual members to autonomously join the community by registering their data into the integration system.

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 [28, 18]. 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-based 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.

The RIDE tool. 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 [28, 18], MS BizTalk Server [5] and Stylus Studio [7]) 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-based integration system enabled by RIDE is shown in Figure 1. In such systems the community initiator starts by designing the target schema. Note

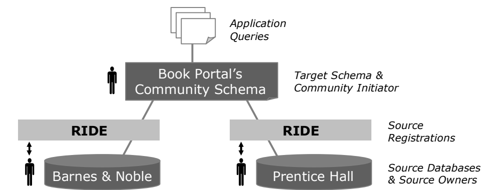


Figure 1: Community-Based Integration Architecture

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 predefined 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 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.

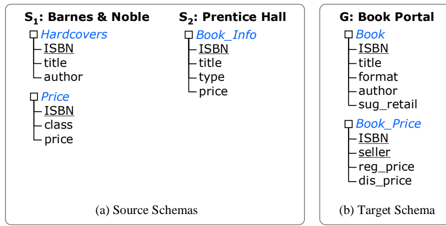
1.1 Contributions

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 5.3. 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.

[1] Ad 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-based integration system can support such evolution aspects the reader is referred to Appendix D.1.

[2] In this paper, 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.



(a) Source Schemas (b) Target Schema

Figure 2: Source schemas and Target schema

```
SELECT title, format, seller, reg_Price, dis_Price
FROM   Book, Book_Price
WHERE  Book.ISBN = Book_Price.ISBN AND Book.author = "Ullman"
```

Figure 3: Application query

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

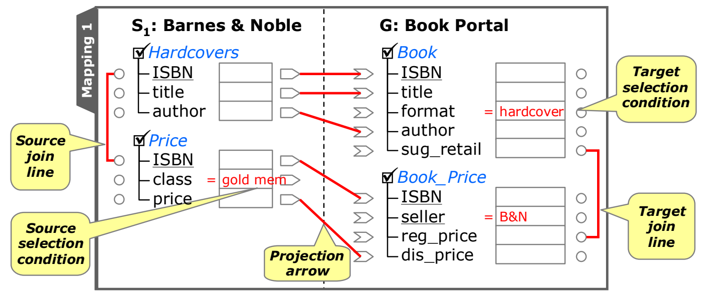


Figure 4: Barnes & Noble Registration

and assertions has been addressed extensively in related work and is beyond the scope of our paper (see related work in Section 7).

Paper outline. We first present RIDE informally, describing our running example in Section 2, traditional schema mapping GUIs in Section 3, and a sample interaction highlighting RIDE's functionality in Section 4. Section 5 formally defines the levels of self-reliance and inconsistency. The algorithms underlying RIDE and their experimental evaluation are described in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2. 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 $\mathcal{G}$, shown in Figure 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 Bibliophilia'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 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 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. For a discussion on RIDE's suggestions for parameterized queries, please refer to Appendix A.

3. MAPPING INTERFACES

RIDE's front end resembles graphical interfaces of schema mapping tools, such as IBM Clio [28, 18], MS BizTalk Server [5] and Stylus Studio [7]. 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 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 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 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 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 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 5.1.

4. 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 5.

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 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 5 and 6 depict the respective screenshots.

4.1 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 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 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. [3]

For example, Snapshot 1.1 shows that to create a self-sufficient registration w.r.t. the query of Figure 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.

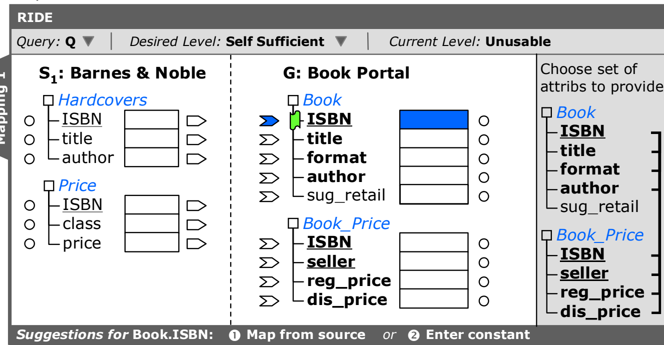
Intra-source assertions. However, if for the sake of contributing to several queries the owner prefers to export more tuples than those relevant to Q, she may choose to not include the selection condition in her mapping. In that case, RIDE will ask her if she believes that the exported tuples will always 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 [6].

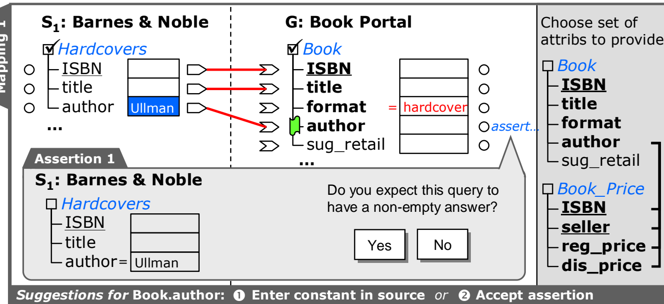
For example, consider Snapshot 1.2 showing the suggestions for

[3] 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.

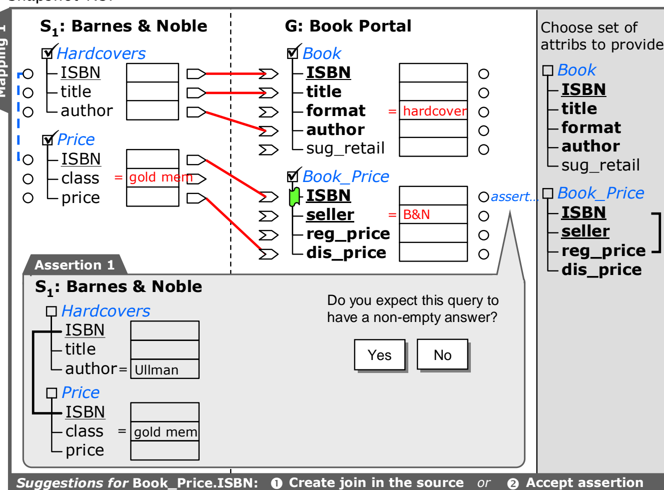
Snapshot 1.1:



Snapshot 1.2:



Snapshot 1.3:



Snapshot 1.4:

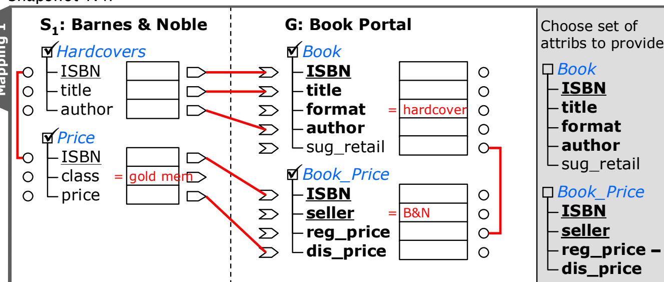


Figure 5: Sample interaction for the B & N registration

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 Ullman (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 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 Price tuples to contribute to the query, the source database needs to contain an Ullman book in table Hardcovers that joins on ISBN with a 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 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 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 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 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 tu-

ples 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.

4.2 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.*

4.3 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.

5. FORMAL SPECIFICATIONS

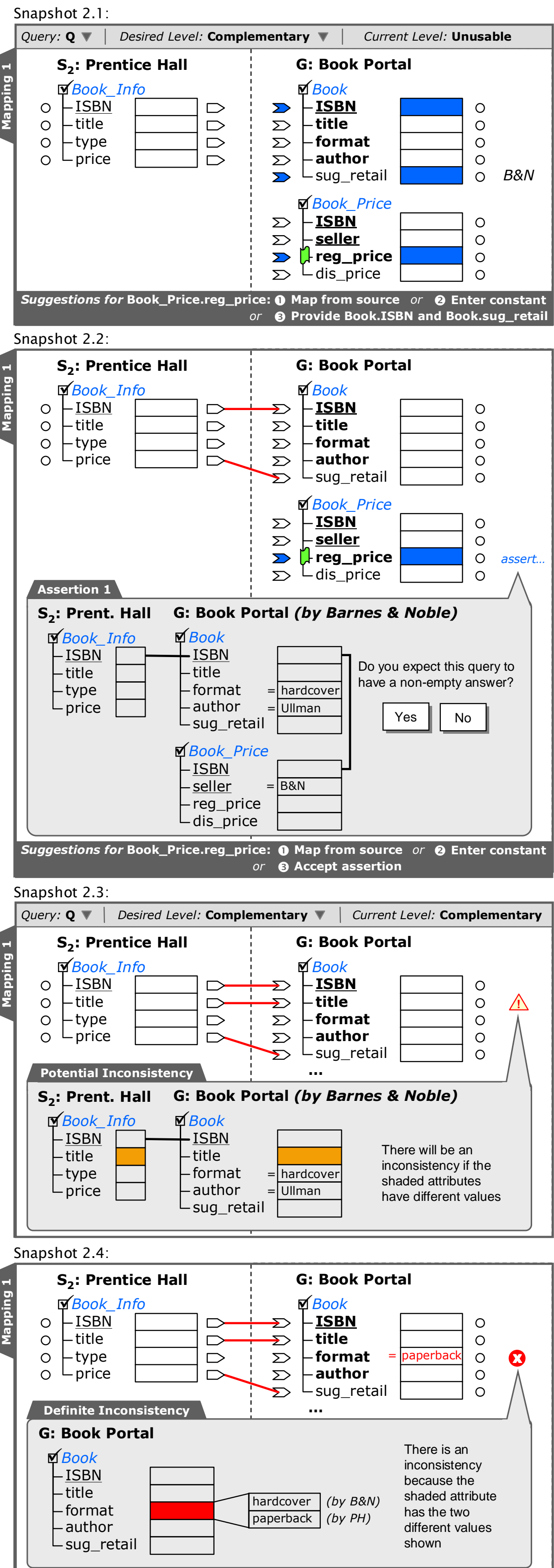


Figure 6: Sample interaction for the Prentice Hall registration

In this section we describe the framework of community-based integration and provide formal definitions for the RIDE concepts described in Section 4. These definitions will be utilized in the description of the algorithms in Section 6.

5.1 Community-based Integration

Community-based 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 [19, 23, 26, 28] 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 [23, 25].

Source and Target Schemas and corresponding Constraints. A community-based integration system integrates a set of source (local) databases with source schemas $\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n$ through a virtual target (global) database over target schema $\mathcal{G}$. 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 [9]. In the following, $\Delta_{\mathcal{S}_i}$ denotes the set of all source constraints over source schema $\mathcal{S}_i$. Similarly $\Delta_{\mathcal{G}}$ represents the set of target constraints. A database instance DB_i over schema $\mathcal{S}_i$ satisfies the set of constraints $\Delta_{\mathcal{S}_i}$, denoted as $DB_i \models \Delta_{\mathcal{S}_i}$ if it satisfies all constraints in the set.

Registrations and Mapping Constraints. The correspondence between a source schema $\mathcal{S}_i$ and the schema $\mathcal{G}$ 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 $\mathcal{S}_i$ and the target schema $\mathcal{G}$, 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* [23, 25].

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 a equality condition with that constant in U (V). *For instance, B&N 's mapping shown in Figure 4 corresponds to the mapping constraint $U \subseteq V$ with U, V given below:*

$$U(I_1, T, A, I_2, P) : -Hardcovers(I_1, T, A), Price(I_2, C, P), \\ I_1 = I_2, C = \text{'}gold_member\text{'}$$

$$V(I_1, T, A, I_2, DP) : -Book(I_1, T, F, A, SR), Book_Price(I_2, S, RP, DP), \\ F = \text{'}hardcover\text{'}, SR = RP, S = \text{'B\&N'}$$

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 paper, 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 [25, 23].

Starting from a set of source instances $\overline{DB} = DB_1, ..., DB_n$ satisfying the source constraints, the set of corresponding GLAV registrations $\bar{R} = R_1, ..., R_n$ does not define a *single* target instance. Instead there is in general a set $Targets_{\bar{R}}(\overline{DB})$ of possible target instances that satisfy the registrations and the target constraints $\Delta_{\mathcal{G}}$:

$Targets_{\bar{R}}(\overline{DB}) = \{G | \bigwedge_{i=1}^{n} (DB_i, G) \models R_i \wedge G \models \Delta_{\mathcal{G}}\}$.

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:

$$CertQ_{\bar{R}}(\overline{DB}) = \bigcap_{G \in Targets_{\bar{R}}(\overline{DB})} Q(G).$$

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 4 and its database DB_1 has general and member price information for an Ullman book stored in the following two tuples: Hardcovers*("5", "DB Systems", "Ullman") and* Price*("5", "gold members", "\$80"). Any target instance that satisfies the source's registration will contain at least two tuples of the form:* Book*("5", "DB Systems", "hardcover", "Ullman", X)* Book_Price*("5", "B&N", X, "\$80"). 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.*

5.2 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 $\mathcal{S}_1, ..., \mathcal{S}_n$, databases $DB_1, \ldots, DB_n$ and corresponding registrations $R_1, ..., R_n$. Let the set of all accepted intra- and inter-source assertions be $\mathcal{A}$ and denote with $\overline{DB} \models \mathcal{A}$ the fact that they are satisfied by the collection of source databases.

Formally, the integration system is potentially inconsistent iff
$\exists DB_1, ..., DB_n$ over $\mathcal{S}_1, ..., \mathcal{S}_n$ s.t.
$DB_i \models \Delta_{\mathcal{S}_i}$ for all $1 \leq i \leq n$, $\overline{DB} \models \mathcal{A}$, 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
$\forall DB_1, ..., DB_n$ over $\mathcal{S}_1, ..., \mathcal{S}_n$
if $DB_i \models \Delta_{\mathcal{S}_i}$ for all $1 \leq i \leq n$ and $\overline{DB} \models \mathcal{A}$, then
$Targets_{R_1,...,R_n}(DB_1, ..., DB_n) = \emptyset$

Examples of both inconsistency kinds were given in Section 4.2.

5.3 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 $\mathcal{A}_{n+1}^{intra}$ and the inter-source assertions involving source $n+1$ be $\mathcal{A}_{n+1}^{inter}$. As above, the collection of all assertions pertaining to sources 1 through n is denoted $\mathcal{A}$.

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 5 by providing an actual value for the regular price through a projection arrow from Price.price*, then B&N's registration would be self-sufficient w.r.t. the query of Figure 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
$\forall DB_{n+1}$ over $\mathcal{S}_{n+1}$ s.t. $DB_{n+1} \models \Delta_{\mathcal{S}_{n+1}}$ and
$DB_{n+1} \models \mathcal{A}_{n+1}^{intra}$: $CertQ_{R_{n+1}}(DB_{n+1}) \neq \emptyset$.

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 *Complementary*. Complementarity is usually enabled by primary key constraints on the target schema.

For example, PH's registration in Snapshot 2.3 of Figure 6 is Complementary w.r.t. the query of our running example and B&N's registration of Snapshot 1.4 in Figure 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 *Complementary* w.r.t. Q and $\bar{R}$ iff
it is not Self Sufficient w.r.t. $\bar{R}$ and
$\forall DB_1, ..., DB_{n+1}$ over $\mathcal{S}_1, ..., \mathcal{S}_{n+1}$
s.t. $DB_i \models \Delta_{\mathcal{S}_i}$ and $\overline{DB} \models \mathcal{A} \cup \mathcal{A}_{n+1}^{inter} \cup \mathcal{A}_{n+1}^{intra}$:
$CertQ_{R_1,...,R_n,R_{n+1}}(DB_1, ..., DB_n, DB_{n+1}) \supsetneq$
$CertQ_{R_1,...,R_n}(DB_1, ..., DB_n)$.

Unusable. Finally, a source registration that is neither self-sufficient nor complementary is called *Unusable*.

6. 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 $CanSource$ and a corresponding possible target instance $CanTarget$. The instance $CanSource$ is over schema $\bigcup_i \mathcal{S}_i$, 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 [9]. While we do not describe the well-known chase in detail here, we show a lesser known algorithm for its implementation [18, 15]: 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. [15].

algorithm Chase (by query evaluation)
input: instance I, set of constraints Δ
output: instance J obtained by modifying I to satisfy Δ
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 $t \in V(J)$

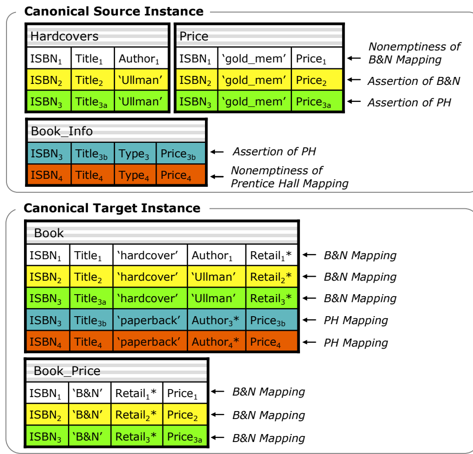


Figure 7: Canonical Source and Target Instance

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 [18, 15].

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:*

$$U_{PK}(I,T_1,F_1,A_1,SR_1,T_2,F_2,A_2,SR_2) : - \\ Book(I,T_1,F_1,A_1,SR_1), Book(I,T_2,F_2,A_2,SR_2)$$
$$V_{PK}(I,T,F,A,SR,T,F,A,SR) : -Book(I,T,F,A,SR)$$

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 $\bigcup_{i \in \mathcal{N}} \mathcal{S}_i \cup \mathcal{G}$
2. for each source query Q appearing in
a mapping constraint or assertion for some source $i \in \mathcal{N}$
3. add a fresh copy of Q's body (one tuple per query atom) to I
4. J := **Chase**$(I, \bigcup_{i \in \mathcal{N}} \Delta_{\mathcal{S}_i} \cup \bigcup_{i \in \mathcal{N}} R_i \cup \Delta_{\mathcal{G}})$
5. $CanSource$:= restriction of J to source relations
$CanTarget$:= 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 $\bigcup_{i \in \mathcal{N}} \Delta_{\mathcal{S}_i}$ 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 $\Delta_{\mathcal{G}}$, 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 5 and 6 respectively, 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 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.*

6.1 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 $1, \ldots, n$
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 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 6.

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

THEOREM 1. *The registrations R_1, ..., R_n lead to definite inconsistency if and only if* **mkCanInst**(*{1, 2, ..., 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 C) 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.

6.2 Deciding Self-Reliance Levels

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

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 $CanTarget$ [8, 25, 18].

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 iff R_{n+1} is Complementary w.r.t. R_1, ..., R_n and Q
begin
$(CanSource, CanTarget) := \mathbf{mkCanInst}(\{1, \ldots, n\})$
$(CanSource', CanTarget') := \mathbf{mkCanInst}(\{1, \ldots, n, n+1\})$
if $Q(CanTarget') \supsetneq Q(CanTarget)$ **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 $R_1, ..., R_n$ 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 $R_1, \ldots, R_{n+1}$.*

6.3 Computing Suggestions

As described in Section 4, 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 4.1). 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:

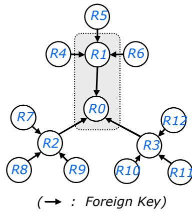
1. Projection arrows and target conditions: RIDE checks if the selected attribute can be provided directly through a projection arrow from some source attribute or through a target condition.

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.

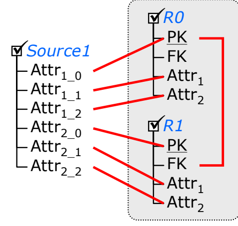
3a. Intra-source assertions due to query selections: If the query contains a constant in the clicked target attribute and this attribute is already mapped into from a source attribute, RIDE generates an assertion that for some source tuple the corresponding attribute value equals the query constant. 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 attribute. 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 7).



(a) Target Schema with d = 2 & r = 3



(b) Registration of source providing R0, R1

Figure 8: Experimental Setting

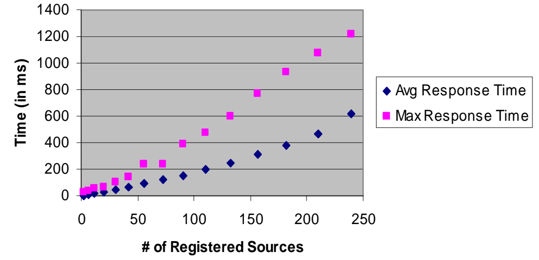


Figure 9: RIDE's response times w.r.t. # of registered sources

These values are known as *labeled nulls* [18]. All occurrences of the same labeled null mark attribute occurrences sharing the same (unspecified) value. RIDE attempts to provide concrete 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.

6.4 Complexity

Termination of the Chase. The property of *weak acyclicity* of a set of constraints is sufficient to guarantee that any chase sequence terminates [18, 16]. 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 [15]. Of course, for arbitrary constraints the chase may not terminate, as termination is undecidable [9].

Complexity of creating the target instance. Since all algorithms involve 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.

Let e be the maximum length (in number of relational atoms) of a source query appearing in any mapping constraint. An analysis of the chase behavior yields that **mkCanInst** runs in worst-case time exponential to e (see Appendix B). However 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 source schema, and more effectively, by the owner's limited capacity of comprehending complex mappings. Indeed, whenever possible, owners prefer to split the registration into many small mappings rather than wielding a single large one.

6.5 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 $\frac{r^{d+1}-1}{r-1}$ tables. The number of sources is $\frac{r^{d+1}-1}{r-1} - 1$ and both their number as well as their overlap increases with d and r. Figure 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 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 av-

erage 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.

7. RELATED WORK

RIDE adopts the GLAV formalism introduced in the context of open-world integration systems [19, 23, 25], later used in data exchange [18] and peer-to-peer integration systems [24, 20]. However, none of these lines of work addresses autonomous source registration, levels of self-reliance, or visual guidance towards them.

The idea of self-reliance was first introduced in [13], but the notions presented here are appropriately adapted to RIDE's needs. In [13], the definitions used *existential* quantification over the source databases, thus checking whether there exist *some* source databases with a desired self-reliance level. This corresponds to *potential* self sufficiency and complementarity, as opposed to the *definite* flavor checked by RIDE. Besides calling for a completely new set of algorithms, the design decision we take here ensures that the self-reliance level holds for the current source databases as well as all their updates compatible with the integrity constraints and assertions. This departure from [13] 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 [22].

Efficient Implementations of the chase algorithm based on query evaluation are reported and evaluated in [15, 18, 12].

Recently, in Cimple/DBLife [17] 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* [4] and *FirstLogic Information Quality* [2] detect primary key violations (so-called duplicate tuples) by inspecting the underlying data instances. Other projects allow inconsistencies but rewrite application queries to take into account only the consistent part of the database [21, 11], or to compute probabilities for each of the inconsistent duplicate tuples [10]. Our focus on inconsistency is complementary, emphasizing prevention and explanation at registration time rather than detection and resolution at run time.

8. 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**.

9. REFERENCES

[1] BIRN: Biomedical Informatics Research Network. http://www.nbirn.net/.

[2] Firstlogic Information Quality. http://www.firstlogic.com/dataquality/.

[3] GEON: Geosciences Network. http://www.geongrid.org/.

[4] IBM WebSphere QualityStage. http://www.ascential.com/products/qualitystage.html.

[5] Microsoft Biztalk Server. http://www.biztalk.org/.

[6] Microsoft SQL Server. http://www.microsoft.com/sql/.

[7] Stylus Studio. http://www.stylusstudio.com/.

[8] S. Abiteboul and O. M. Duschka. Complexity of Answering Queries Using Materialized Views. In *PODS*, 1998.

[9] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[10] P. Andritsos, A. Fuxman, and R. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.

[11] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[12] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *CAiSE*, 2002.

[13] A. Deutsch, Y. Katsis, and Y. Papakonstantinou. Determining source contribution in integration systems. In *PODS*, 2005.

[14] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[15] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[16] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *ICDT*, pages 225–241, 2003.

[17] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Eng. Bull.*, 29(1):64–72, 2006.

[18] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[19] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *AAAI*, 1999.

[20] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W.-C. Tan. Peer data exchange. In *PODS*, 2005.

[21] A. Fuxman and R. Miller. First-order query rewriting for inconsistent databases. In *ICDT*, pages 337–351, 2005.

[22] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *PODS*, 1994.

[23] A. Halevy. Logic-based techniques in data integration. In *Logic Based Artificial Intelligence*, 2000.

[24] A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer-data management system. *TKDE*, 2004.

[25] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[26] J. D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[27] Y. Velegrakis, J. Miller, and L. Popa. Preserving mapping consistency under schema changes. *VLDB Journal*, 13(3):274–293, 2004.

[28] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD*, 2004.

[29] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, pages 1006–1017, 2005.

APPENDIX

This appendix contains various discussions omitted from the main body of the paper due to lack of space. Section A discusses RIDE's suggestions in the presence of parameterized queries. Section B extends Section 6.4 by providing a more detailed complexity analysis of the algorithm **mkCanInst** for the generation of the canonical source and target instances. Section C contains the proof of Theorem 2 (i.e. of the undecidability of potential inconsistency). Finally Section D discusses two possible extensions of RIDE and the community-based framework: First, an extension allowing the community-based integration framework to support evolution of the target schema within a single community or coalescing of multiple communities that have emerged on the same topic into a larger community. Second, an extension of RIDE to allow contribution to a list of queries.

A. RIDE'S SUGGESTIONS FOR PARAMETERIZED QUERIES

Throughout the paper we employed a single non-parameterized 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 in Section 4 author was a parameter, RIDE would make the same suggestions apart from the ones shown in Snapshot 1.2 of Figure 5 (while removing 'Ullman' from all others).

All definitions and algorithms can be straightforwardly extended to parameterized queries.

B. EXTENDED COMPLEXITY ANALYSIS

In this section we show the exact complexity of the algorithm **mkCanInst**, which we omitted from Section 6.4 due to lack of space.

As in the aforementioned Section we refer to the typical case in which the integrity constraints are (weakly acyclic) sets of primary keys, the target schema (and constraints) are fixed and only the source schemas and their registrations vary.

We introduce the following notation: $N_{\mathcal{S}}$ 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 **mkCanInst** runs in worst-case time $O(N_{\mathcal{S}} b^e t + N_v k^2)$. 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 [9]. 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.

C. PROOF OF UNDECIDABILITY FOR POTENTIAL INCONSISTENCY

- **Source schema** $\mathcal{S} = \{E_{\mathcal{S}}$: 3-ary $\}$
- **Target schema** $\mathcal{G} = \{C$: 5-ary, R: 2-ary$\}$
- **Set of assertions** $\mathcal{A} = \emptyset$
- **Set of source constraints** $\Delta_{\mathcal{S}} = \{(U^i_\delta \subseteq V^i_\delta) | 1 \leq i \leq 2\}$
  - $U^1_\delta(s, l_1, t_1, l_2, t_2)$:- $E_{\mathcal{S}}(s, l_1, t_1), E_{\mathcal{S}}(s, l_2, t_2)$
  - $V^1_\delta(s, l, t, l, t)$:- $E_{\mathcal{S}}(s, l, t)$
  - $U^2_\delta(t, s_1, l_1, s_2, l_2)$:- $E_{\mathcal{S}}(s_1, l_1, t), E_{\mathcal{S}}(s_2, l_2, t)$
  - $V^2_\delta(t, s, l, s, l)$:- $E_{\mathcal{S}}(s, l, t)$
- **Set of target constraints** $\Delta_{\mathcal{G}} = \{(U^3_\delta \subseteq V^3_\delta)\}$
  - $U^3_\delta(x', y')$:- $R(x, y), C(x, y, i, x', y')$
  - $V^3_\delta(x', y')$:- $R(x', y')$
- **Set of mappings** $M = \{(U^i_{\mathcal{G}} \subseteq V^i_{\mathcal{G}}) | 1 \leq i \leq 2n\}$

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

  $U^i_{\mathcal{G}}(x_1, y_1, i, x_{k+1}, y_{l+1})$:-
  $E_{\mathcal{S}}(x_1, a_1, x_2), E_{\mathcal{S}}(x_2, a_2, x_3), ..., E_{\mathcal{S}}(x_k, a_k, x_{k+1}),$
  $E_{\mathcal{S}}(y_1, b_1, y_2), E_{\mathcal{S}}(y_2, b_2, y_3), ..., E_{\mathcal{S}}(y_l, b_l, y_{l+1})$

  $V^i_{\mathcal{G}}(x_1, y_1, i, x_2, y_2)$:- $C(x_1, y_1, i, x_2, y_2)$

  let l_0 be a letter not in Σ

  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

  $U^{i+n}_{\mathcal{G}}(x_k, y_l)$:-
  $E_{\mathcal{S}}(s_0, l_0, s)$,
  $E_{\mathcal{S}}(s, a_1, x_1), E_{\mathcal{S}}(x_1, a_2, x_2), ..., E_{\mathcal{S}}(x_{k-1}, a_k, x_k),$
  $E_{\mathcal{S}}(s, b_1, y_1), E_{\mathcal{S}}(y_1, b_2, y_2), ..., E_{\mathcal{S}}(y_{l-1}, b_l, y_l)$

  $V^{i+n}_{\mathcal{G}}(x_1, y_1)$:- $R(x_1, y_1)$

  else remove the constraint $(U^{i+n}_{\mathcal{G}} \subseteq V^{i+n}_{\mathcal{G}})$

Figure 10: Auxiliary Integration System IS_{aux}

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 Σ (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 $\mathcal{S}$, a target schema $\mathcal{G}$, a set of assertions $\mathcal{A}$, a set of source constraints $\Delta_{\mathcal{S}}$, a set of target constraints $\Delta_{\mathcal{G}}$ and a set of mapping constraints M) s.t. a PCP instance has a solution iff the corresponding integration system is potentially inconsistent.

Same as IS_{aux} shown in Figure 10 with an additional target relation I and two additional target constraints involving I:

- **Source schema** $\mathcal{S} = \{E_{\mathcal{S}}$: 3-ary $\}$
- **Target schema** $\mathcal{G} = \{C$: 5-ary, R: 2-ary, I: 2-ary $\}$
- **Set of assertions** $\mathcal{A} = \emptyset$
- **Set of source constraints** $\Delta_{\mathcal{S}} = \{(U^i_\delta \subseteq V^i_\delta) | 1 \le i \le 2\}$
- **Set of target constraints** $\Delta_{\mathcal{G}} = \{(U^i_\delta \subseteq V^i_\delta) | 3 \le i \le 5\}$
  $U^4_\delta(x)$:- $R(x, x)$
  $V^4_\delta(x)$:- $I(x, 1), I(x, 2)$

  $U^5_\delta(x, y_1, y_2)$:- $I(x, y_1), I(x, y_2)$
  $V^5_\delta(x, y, y)$:- $I(x, y)$
- **Set of mappings** $M = \{(U^i_{\mathcal{G}} \subseteq V^i_{\mathcal{G}}) | 1 \le i \le 2n\}$

Figure 11: Integration System IS used in the Reduction

For ease of exposition we first create an auxiliary integration system IS_{aux} shown in Figure 10. Then we extend it to the actual integration system IS used in the reduction as shown in Figure 11. 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 $\mathcal{S}$ and a single registration M.

Let us first present IS_{aux}. Source relation $E_{\mathcal{S}}$ is the edge relation of a labeled directed graph with $E_{\mathcal{S}}(s, l, t)$ 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 $E_{\mathcal{S}}(x_1, a_1, x_2), ..., E_{\mathcal{S}}(x_p, a_p, x_{p+1})$. 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 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 $(U^i_\delta \subseteq V^i_\delta), 1 \le i \le 2$ 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 $(U^i_{\mathcal{G}} \subseteq V^i_{\mathcal{G}})$, $1 \le i \le n$ are used to capture the intended meaning for relation C.

Target constraint $(U^3_\delta \subseteq V^3_\delta)$ and mapping constraints $(U^i_{\mathcal{G}} \subseteq V^i_{\mathcal{G}}), n+1 \le i \le 2n$ 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 $(U^3_\delta \subseteq V^3_\delta)$ and the base case of the recursion is captured by constraints $(U^i_{\mathcal{G}} \subseteq V^i_{\mathcal{G}}), n+1 \le i \le 2n$. 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 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 $(U^i_\delta \subseteq V^i_\delta), 4 \le i \le 5$ 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 $R(x, x)$.

D. EXTENSIONS/FUTURE WORK

D.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 [27, 29, 14] 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 [27, 29]. 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 [14]). 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.

D.2 Contributing to a List of Queries

In this paper, 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 vis-

ited 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.