Managing Schema Mappings in Highly Heterogeneous Environments

by

Yannis Velegrakis

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

.

Abstract

Managing Schema Mappings in Highly Heterogeneous Environments

Yannis Velegrakis

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2005

Integration, transformation, and translation of data is increasingly important for modern information systems and e-commerce applications. Views, and more generally, transformation specifications, or mappings, provide the foundation for many data transformation applications.

Mappings are usually specified manually by data administrators that are familiar with the semantics of the data and have a good knowledge of the transformation language. The task of generating and managing mappings is laborious, time consuming and error-prone since data administrators are called on to write complex mappings in which they specify in tedious detail how the data is to be transformed. Even once deployed, mappings must remain under constant supervision since changes in the structure of the data may require changes in the mappings.

In this dissertation, we elaborate on the development of mapping management tools that are intended to shield administrators from the laborious task of mapping management. In particular, we present a novel framework for generating mappings between any

combination of XML and relational schemas. A set of high-level binary relationships between the elements of the two schemas, which are specified by a user or generated by a tool, are combined together to form semantically meaningful mappings. These mappings are guaranteed to be consistent with the constraints of the schemas. To handle schemas that are dynamically modified, we describe a methodology for automatically detecting the mappings that have become invalid as a result of a schema change and rewriting them to become consistent with the modified schema. Each rewriting is generated in a way that preserves, as much as possible, the semantics of the initial mapping. Finally, we show how collections of schemas and mappings can be used in queries to provide a better understanding of how data has been integrated and transformed.

.

Dedicated to my brother Niko

As you set out for Ithaca
hope your road is a long one,
full of adventure, full of discovery.
...

Keep Ithaca always in your mind.
Arriving there is what you're destined for.
But don't hurry the journey at all.
Better if it lasts for years,
so you're old by the time you reach the island,
wealthy with all you've gained on the way,
not expecting Ithaca to make you rich.

Ithaca gave you the marvelous journey.
Without her you wouldn't have set out.
She has nothing left to give you now.
And if you find her poor, Ithaca won't have fooled you.
Wise as you will have become, so full of experience,
you'll have understood by then what these Ithakas mean.

From Kostantinos Kavafis' Ithaca

Acknowledgements

I am indebted to my supervisors John Mylopoulos and Renée J. Miller. This dissertation would never have been possible without their guidance and support. John's invaluable experience in many different research fields taught me to see the big picture and how to value things by looking behind the pure database perspective. Renée, on the other hand, taught me how to look at the details and how to investigate the foundations of the problems. She taught me how to present my results in a complete, accurate and professional way. I was really lucky to have such great supervisors.

I would like to thank the other members of my PhD committee, Ken Sevcic, Alberto Mendelzon, Nick Koudas and Eric Yu for their continuous and valuable feedback during the various phases of this thesis. I am also grateful to my external appraisal H.V. Jagadish the many interesting comments.

I cannot end this thesis without saying a very special thank you to Lucian Popa. Lucian has been a third supervisor for me. He taught me how large projects are implemented and how to combine research and development. Close to him, I learned how to work efficiently and how to ensure that my work is always complete and accurate. This thesis may not have been in its current complete form without his guidance and support. Apart from a great collaborator, Lucian has also been a great friend.

This thesis has been supported by both academia and industry. I thank Kathy, Marina, Joan, Kamran, Simon, Samir and Linda for the excellent collaboration, and the Department of Computer Science at the University of Toronto for its support. I thank

IBM Canada for providing me with a great working place in the Toronto Lab. Special thanks to Marin Litoiu, Kelly Lyons and the IBM CAS members for bringing me in contact with product developers and real users. I also thank them for their support through the IBM CAS fellowship. I am also grateful to IBM, for the IBM PhD fellowship. I am grateful to IBM Almaden Research Center and the whole Clio team for the opportunity I was given to participate in state-of-the-art research through my two summer internships.

I have no words to thank my parents Fofo and Manoli for their love and support. If I managed to get in the PhD program is because of them. Finishing this dissertation, is an indication that their patience and sacrifices were not for nothing. A deep thank goes to Cristiana, my partner and friend, for her love, support, patience and faith in me. I am dedicating this thesis to a person that makes me very proud, my brother Niko.

During my stay in Toronto, I was fortunate to meet great people and make good friends that made me feel like home. A special thank goes to Perikli and Taso, for putting up with my strange ideas for six years. I believe that the time we spent together in research and social life will keep us friends for ever. I also feel the need to thank them, along with Vasso, for helping me survive the hard year of 2001. I am grateful to all the members of the grspam mailing list for our online and offline conversations, as well as our great parties at Dalton's place. I will never forget Panayiotis' long scarf, Giakoupis' love for tuna, Periklis' tiny portion lunches, Tasos' complaints about everything, Rozalia's sweat smile, Themis' love for lentils, Anastasia's passion for Kinder eggs, Katerina's cooking habits, Stavros' dancing energy, Fanis' distinct laugh, Katsirelos' big belly contest, Nicolas' $500 car, and many other that I have no space to mention. I may be leaving Toronto, but they will all stay in my heart for ever.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

We are witnessing a proliferation of independently developed data sources that are heterogeneous in their design and content. The ability to exchange information contained in such sources is a fundamental requirement for modern information systems. Such systems use mappings to translate data from one format to another. In this dissertation, we present techniques and tools for automating the tasks of generating, maintaining and querying mappings.

The following section introduces the schema mapping problem and motivates the need for tool support. Section 1.2 discusses the key research issues raised in the development of tools for managing mappings. Section 1.3 presents a number of applications and environments in which mappings are used extensively. Finally, Section 1.4 outlines the organization of the rest of the dissertation.

1.1 Motivation

Large amounts of information are currently freely available on the Web in a number of autonomous sources, stored under different formats, e.g., relational, object-oriented, semi-structured, etc. Modern information systems and e-commerce applications are usually limited to handle information of only a specific structure, e.g., data conforming to a

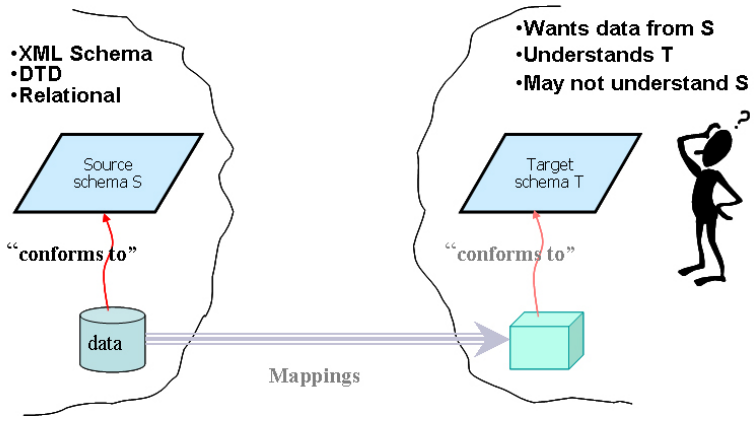


Figure 1.1: Need for data translation

specific relational schema. For such applications, the ability to translate data from one format to another is a fundamental requirement in order to achieve interoperability.

This work concentrates on mappings that permit the translation of data conforming to a given schema, referred to as the *source* schema, into data conforming to another schema, referred to as the *target* schema. Consider, for example, the user in Figure 1.1 who needs to retrieve some data that conforms to a source schema S. Schema S may be a relational schema, a DTD, or an XML Schema. Unfortunately, that user may not completely understand schema S or she may not be permitted to directly query it. Instead, she may understand and be able to query a different target schema T. To retrieve the information of interest, the data of schema S has to be transformed to conform to schema T. The process of generating such transformation functions, or programs, is called *schema mapping* [MHH00], and the functions, or programs used, are called *mappings*. A well-known form of a mapping is a view definition.

Schema mapping is a different problem from schema integration. The goal of schema integration is the creation of a global integrated schema that reflects the information of a number of data sources [BLN86]. Since schema integration is mainly a design problem, the mappings between the individual sources and the integrated schema are explicitly specified by the user or the tool that designs the integrated schema. In schema mapping,

both schemas may have been designed in advance independently of each other and with different requirements in mind. As such, they may describe different data semantics. The problem in schema mapping is then to find the transformation functions (mappings) that describe the relationship between the data represented by the two schemas (Figure 1.2).

Data translation is usually achieved by specialized tools that have the translation rules embedded in their code. The development of such programs requires a lot of effort and is both time consuming and error-prone, since developers may have to understand intricate technical details of many extremely complex translations. There are few tools for managing translation functions. Procedural translations, e.g., transformation procedures in a programming language, may need to be recompiled each time a change in the translations or data is required. Data translations may also be specified in a declarative, data-independent way. Still, data administrators must write and manage complex declarative mappings in which they specify in tedious detail how the data is to be transformed. Furthermore, with the data sources in constant evolution, even once deployed, the translation functions must remain under continuous supervision and be updated manually every time the logical structure or schema of the data changes. Engineers need to establish and reason about the quality and properties of the established translations to ensure the correctness of the transformation. To achieve that, they may need to manually browse large numbers of such function definitions in order to discover those that satisfy

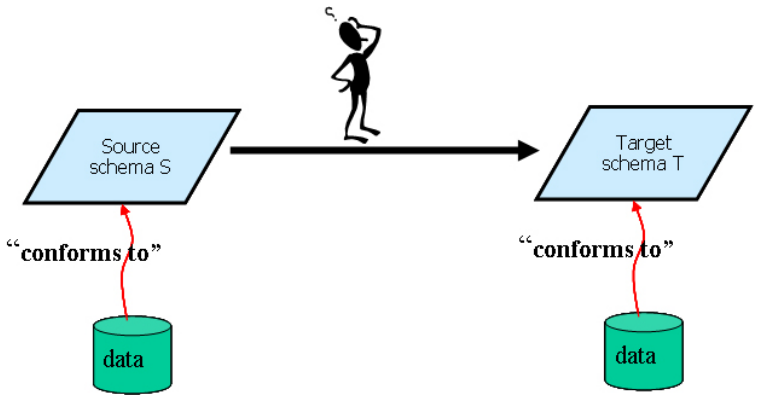


Figure 1.2: Schema mapping

a specific requirement or have a specific property.

For small schemas that use a few simple mappings, manually managing the mappings is a feasible solution. However, as large complicated schemas are becoming prevalent, the complexity and the number of mappings is increasing and manual management is becoming impractical. One soon feels the need for tool support. The goal and contribution of this work is the development of a comprehensive framework and set of tools for generating, maintaining and querying mappings. The ultimate goal is to simplify heterogeneous data translation and make schemas and mappings easier to manage with far less human effort.

This work can be seen within a general framework of *model management* [BLP00] in which schemas and views or mappings are considered and manipulated, using well-defined operators, as first-class citizens of a data management system. In particular, our proposed framework can be seen as a realization of three operators for mappings: *generate*, *update* and *query*. A first-class citizen is an entity that can be used without restrictions, and its existence does not depend on other entities.

1.2 Research Issues and Contributions

The design and implementation of systems for managing mappings raises a number of research issues. Among them, there are three that we address in this work.

Mapping Generation: To shield users from the laborious task of manually generating complex mappings and to reduce the risk of errors, we need to raise the abstraction level of the mappings. Users should not be required to be experts in the transformation language in order to specify a transformation, and should not have to describe the mappings in every detail. They should give a high-level intuitive description of what the mapping should be and the generation of the actual translation program should be automated as much as possible.

In this work, we have developed a mapping generation framework [PHV+02] that permits a user to provide a high-level specification of the mappings between two schemas through a set of binary relationships between the parts of the two schemas. We refer to these relationships as *correspondences*. Correspondences are simple to understand and easy to define. Hence, even users unfamiliar with the complex structure of the schema and the transformation language can use them. These high-level and some times ambiguous correspondences are then used to create data transformations (mappings) which can be used to transform an instance of the source schema to into an instance of the target schema. The two schemas may be any combination of relational or XML Schemas [W3C01], and the generated mappings may be expressed as SQL queries, XQueries [CCDF+01], or XSLT scripts [Cla99].

The process of mapping generation [PVM+02b] we present in this dissertation uses information that is embodied in the schema. In particular, it uses information about the structure of the data described by the schema (schema structure), and the conditions that the data needs to satisfy (schema constraints). Schema structure and constraints have a lot to offer since they embody large portions of the data semantics. While research on data transformation has been very extensive, it has concentrated mainly on the data itself. Schema information has not been considered systematically. As a consequence, the vast potential of schemas, in guiding mapping creation, has so far remained largely untapped. Manual generation of mappings may not take into consideration constraints of the schemas. Our work is the first to consider schema mapping for XML schemas, the first to use nested referential constraints to understand the intended semantics of the transformations, and the first to guarantee that the transformed data will not violate any referential constraints of the target schema. Furthermore, we propose new solutions for deriving new data values that may be required to maintain the semantics of the data after the transformations. The algorithm we present is the heart of the research prototype mapping creation system Clio [MHH+01], that features a graphical interface

and a mining tool to assist the user in generating mappings between two schemas. Our contribution to the Clio project was the development of the core mapping generation algorithm for XML Schemas and nested constraints.

Mapping Maintenance: In dynamic environments like the Web, data sources may change not only their data, but also their schema structure and semantics. Such changes must be reflected in the mappings. Mappings left inconsistent by a schema change have to be detected and adapted accordingly. Clearly, manually maintaining many mappings (even simple mappings like view definitions) for large complicated schemas is impractical. This dissertation proposes a novel framework [VMP03b] for automatically finding rewritings of mappings as schemas evolve. Here again, constraints play an important role in helping to realize the semantics of the schema changes and produce rewritings of the mappings that become inconsistent due to these changes. This is the first work to consider changes not only to the structure of the schema, but also to its semantics, i.e., constraint changes. We consider the problem of how mappings are affected when constraints are modified or schema elements reorganized. We refer to this problem as *mapping adaptation*. One can ill-afford to recreate the mappings from scratch after schemas have been modified. It is advisable, instead, to reuse existing mappings to do the changes incrementally. The reason is that, although mapping creation may be aided tremendously by mapping generation tools, it still requires some input from human experts. It is the semantic decisions input by these experts that our mapping adaptation solution explicitly models and tries to preserve in order to save the most precious administrative resource, human time. To do that, we exploit schema structure, constraints and existing mappings. To help the user even further, we introduce a way of ranking the candidate rewrittings according to how similar they are to the original mapping that is to be updated [VMP04]. The framework has been developed and implemented at the University of Toronto in the mapping maintenance tool ToMAS [VMPM04].

Querying over Mappings: Modern information systems often store data that has

been transformed and integrated from a variety of sources. A common schema is used to facilitate querying. However, such integrated and transparent access to the data may obscure the original source semantics of data items. For many tasks, including data cleaning, it is important to be able to determine not only from where data items originated, but also why they appear in the integration as they do and through what transformation they were derived. Finding the origin of a data element is the problem of data provenance. Data provenance [CW00, BKT01] has been studied at the data level by detecting specific data values in the sources that are responsible for the appearance of a data value in the integration. In this dissertation, we investigate the same problem but at the schema and mapping level by considering how one can declaratively describe the origin of data. Our goal is to be able to answer questions such as "what schema elements in the source(s) contributed to this value?" and "through what transformations or mappings was this value derived?". To this end, we elevate schemas and mappings to first-class citizens of the data repositories by storing and associating them with the actual data. We also design a language (MXQL) that supports queries over both data and meta-data (schemas and mappings) [VMM05]. The same language can also be used by a data administrator to find parts of the schemas or mappings that have a specific property, or are related to some other mapping or schema part. This allows batch processing of mappings instead of the object-at-a-time processing that would otherwise be required for the same task. The goal is to eliminate the gap between data and meta-data by storing and querying meta-data in the same way regular data is stored and queried. This is the first work to allow queries not only on the provenance of the data, but also on the transformations through which the data has been derived.

1.3 Application Domains

Mappings are used extensively in a large number of applications. The increasing complexity of mappings and schemas calls for efficient mapping management solutions. Below, we describe how the mapping management techniques on which we describe in this thesis can contribute in these applications.

Mapping Generation and Mapping Maintenance can be used in e-commerce applications, software engineering tools, for schema evolution, in modeling source descriptions, and in physical design tools.

- **Schema Evolution**. Databases frequently have long lives. During a database's lifetime, the database schema is likely to undergo significant change as new demands are placed on the data (e.g., new kinds of data need to be stored), new data semantics needs to be reflected in the schema, or data needs to be reorganized to improve performance [Ler00]. When the schema of the database is to be updated, the existing data instance needs to be modified to conform to the new schema. This can be done automatically if it is supported by the data management system [Bre96] and the changes are simple [FL96, Nav80]. For more complicated changes or for systems that do not support such a functionality, a solution is to generate a new database that conforms to the new schema, and transfer all the data from the old database to the new one. This requires the generation of a number of mappings from the old to the new schema to describe what parts of the instance data are to be transformed, and how this will be achieved. When schemas are large and the new schemas are dramatically different from the old, mappings may be large and complicated, in which case any tool support for data administrators will be of great help.

- **Software Engineering & Code Migration**. For large legacy systems, there is usually a need for porting old program code to new modern programming lan-

guages and systems. During this porting, it is essential to minimize information loss to avoid unwanted consequences later in the program execution. In this task, the existence of tools that support programmers in generating scripts to perform code translation is important. Furthermore, many software engineering and reverse engineering tools use different formats to represent and store information they have processed. Interoperability among such tools is an important problem since it allows results generated by one tool to be used in another. There has been a lot of effort in the last few years in the development of a standard exchange format for such a purpose, i.e., GLX [Win02]. Unfortunately, a format, even if it is universally adopted, is unlikely to be used by all tools internally. This means that tools will still need to map their data from their own native schema or structure to the one described in the universal format.

- **Schema Integration**. In schema integration, a unified view of a set of heterogeneous data sources is created [BLN86]. Numerous algorithms and tools have been proposed to automate or semi-automate schema integration [SP94]. However, at its core, schema integration is a schema design problem. Some integration choices will necessarily be subjective and different users or designers may wish to make different choices or alter a heuristic choice made by a tool. Some tools anticipate this and, for a limited set of alternative designs, will still produce a correct mapping between the source schemas and the selected integrated schema [SP94]. Others will permit users to use a set of schema transformation operators that can be composed to produce an integrated (transformed) schema [GLS95]. However, these approaches in general do not permit arbitrary changes to the integrated schema. Even a simple horizontal decomposition of an integrated table based on a user-defined predicate will typically require the designer to manually edit the mapping. Furthermore, changes in the source schema (even modest ones) are not supported. Such changes require the integration algorithm to be rerun.

- **Modeling Source Descriptions**. An important and influential proposal in data integration was to define a collection of heterogeneous data sources as views over an integrated schema [LRO96]. This technique (now known as local-as-view or LAV) proved to provide considerable flexibility and data independence for applications on the integrated data (for example, data warehousing applications). LAV can be contrasted with classical data integration where the integrated (global) schema is modeled as a view over the sources (this approach is called global-as-view or GAV). However, in both GAV and LAV systems, schemas have been assumed to be static [GMPQ+97, LRO96]. If a schema change occurs then the mappings need to be redefined.

- **Data Exchange**. In data exchange, mappings are used to transform an instance of a source schema into an instance of a different target schema [FKMP03a]. The source and target schemas may be inconsistent, so that for a given source instance, there may be no target instance that represents the same information. While there are algorithms for detecting large classes of such inconsistencies, designers may wish to modify either the source or target schema to make them consistent. This may be done by cleaning inconsistent data in the source and adding a constraint to the source schema (or modifying its structure to prevent new inconsistencies), or by modifying the target. Efficiently and effectively adapting a mapping to such new constraints or structure modifications (in either the source or target) has not yet been considered.

- **Physical data design**. Physical storage wizards, which permit the customization of physical schemas and storage structures, must maintain a mapping between the physical and logical schemas. A common example of such wizards are tools for customizing the relational storage of XML data [BFH+02]. Such tools evaluate (or help a designer to evaluate) the relative cost of different physical relational

designs. However, they consider only a fixed set of physical schemas, each with a built-in mapping to the given logical schema. To permit a designer to suggest schema designs outside of this limited set, the tool would have to be able to adapt the logical to physical schema mapping to the *ad hoc* user-proposed schema change. Chapter 5 contains a case study on how our framework can contribute in this area.

Querying over mappings is useful for applications in which mapping information needs to be used along with the regular data. This includes, but is not limited to, scientific databases, information portals, data warehouses and peer-to-peer systems.

- **Scientific Databases:** Scientific data available on the Web may be retrieved by third parties, transformed and stored in local databases. For these databases, it is essential for anyone interested in the accuracy and timeliness of the data to know the original location from which a given piece of data was drawn, and the processing that has been applied to it.

- **Information Portals:** Information portals are used to collect and organize information for presentation to users in a consistent way. Knowing the sources and the transformations through which the portal data was derived is important in order to assess its quality since some of the originating sources may be known to contain inaccurate or out-of-date information, and some transformations may not accurately reflect the expected portal semantics.

- **Data Warehouses:** In data warehouses, views over source data are defined, computed, and stored in the warehouse to answer queries about the source data in an integrated and efficient way. Online analytical processing and data mining operations operate on the data to perform analysis and predictions. To support "what-if" scenarios, queries could be parameterized with conditions on meta-data that control how the warehouse instance has been generated.

- **Peer-to-Peer Systems:** In peer-to-peer systems, like Piazza [TIM+03], data passes through numerous peers before reaching the peer that requested that information. During its transfer, data goes through a number of transformations that reflect the way each peer views its acquaintances. In these systems where there is neither a global schema nor a centralized authority, it is easy for an answer set to be contaminated with irrelevant data. Information on how, why and from where each data element appeared in the answer set may help detect and eliminate such irrelevant data.

1.4 Dissertation Organization

Chapter 2 discusses prior work in fields related our work. The focus is on high-level similarities and differences with our approach. We start by describing the general area of data integration, and then we explain how schema mapping provides a new challenge for which one can exploit many of the results in data integration in order to provide efficient and effective solutions. We then describe the subtle difference between mapping adaptation and its related fields. Finally, we present preliminary steps that have been taken in the research community for meta-data management, and explain how we contribute in that area.

Chapter 3 introduces the problem of schema mapping and defines the nested relational data model that is used as a common platform to which relational and XML-schemas are translated. The theory described in the subsequent chapters is also based on this model. In addition, the chapter describes how the translation from the various models to the nested relational model is performed and how the type checking mechanism works.

Chapter 4 is divided in two main parts. The first describes how from a given set of high-level descriptions of how the elements of two schemas relate to each other, a tool can produce a set of mappings that are consistent with this high-level description [PVM+02b].

This is the process of *schema mapping*. The second part deals with the problem of *Data Exchange*, that is, given two schemas, a mapping between them and an instance of the first schema specify how to compute an instance of the second schema as described by the mapping. The chapter also describes how the solutions of these two parts were implemented and used in the mapping generation tool Clio [PHV+02].

Chapter 5 deals with the problem of *mapping adaptation*, which is how to maintain the consistency of mappings that have been defined between two schemas while these schemas evolve. A set of primitive schema transformations are considered and, for each transformation, we explain how the mappings are modified. Then the implementation of this framework in a tool, called ToMAS, is described and our experience with that tool is presented [VMP03b, VMP04, VMPM04].

Chapter 6 describes how schemas and mappings can be considered first-class citizens of a repository and a query language. This allows the management of data and meta-data in a unified way. It also allows meta-data about the schema and the mapping to be used in queries. At the end, our experience with the implementation and application of the specific framework in a real world scenario is described [VMM05].

The thesis concludes in Chapter 7 with a description of the main areas of contribution and a short list of the future directions that can be taken as a continuation of this work.

Chapter 2

Related Work

In this chapter, we discuss prior work on topics related to this dissertation. In Sections 2.1 and 2.2, we summarize work in the area of data and schema integration, and we indicate how it differs from our work on schema mapping that will be presented in Chapter 4. Section 2.3 discusses work done on data translation and shows how it complements our work on schema mapping. Section 2.4 describes the data exchange problem and explains how it differs from data integration. Section 2.5 describes work on view maintenance and other related fields. The discussion clarifies why mapping maintenance introduced in Chapter 5 is a new and interesting challenge. Finally, Section 2.6 discusses recent advances in the area of meta-data management. Meta-data management is a broad area that has motivated our work of Chapter 6 on querying meta-data.

2.1 Data Integration

The research literature contains a substantial body of work on the topic of data integration. Data integration deals with the problem of providing an integrated view of data that is distributed in a number of sources. Based on the location of the integrated data, Data Integration Systems (DIS) can be broadly classified in two approaches: the *materialized* [Wid95] (or *warehouse*) approach and the *virtual* [Wie92] approach.

In a *materialized* approach, data originating from different sources are integrated and stored in one single new database. Queries are performed on this integrated database, while the various sources continue to operate as before. The assumption is that all the information of interest has been integrated before any user query is executed. The advantage of this approach is that information can be integrated, translated, and filtered in advance, so it will be immediately available when requested. Moreover, integrated data can be locally modified without affecting the sources, and can contain historical information. The drawback is that there is a cost for keeping the information up-to-date. This makes the *materialized* approach appropriate for systems that require high performance of queries on aggregated, pre-processed, and annotated data.

In the *virtual* approach, the data remains in its native sources. An intermediate layer, called *middleware*, receives the user's query, analyzes it and sends the appropriate query requests to the corresponding sources. The results are retrieved, integrated, sometimes processed even further, and sent back to the user. The advantage of this approach is that no data is replicated, and the answers to the queries are always up-to-date. However, query answering takes longer, and answers may be incomplete due to source failures or network delays. Hence, the *virtual* approach is more appropriate for systems with large number of sources, frequently changing data, no critical response times, and with a great variety of user query needs.

The goal of data integration is to shield the end user from the heterogeneity of the information, the multiple syntactic discrepancies, the different data models and query languages, the various access methods of the individual data sources, as well as the location of data. It aims to provide the user with a uniform data representation and access method. One of the first architectures that was suggested was the construction and use of a *global schema* as indicated in Figure 2.1(a). Intuitively, a *schema* is a set of symbols and expressions over a specific alphabet that describe the structure of the data. A common data model is used in order to avoid mapping data among all

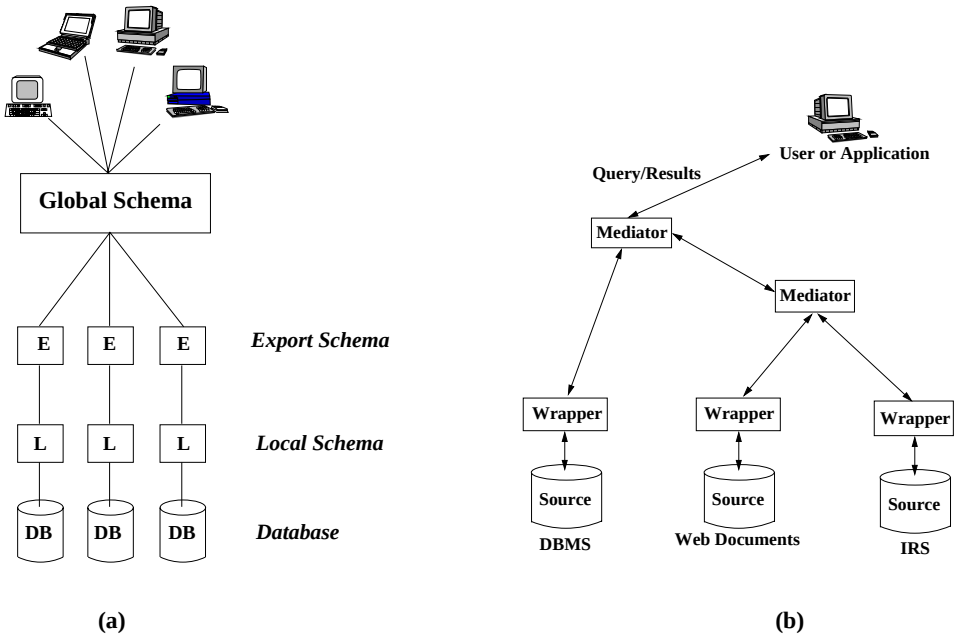


Figure 2.1: Integration architectures

the different data models at run time. The local schema of each database is translated into a component schema which represents the same information but is expressed in the common data model. This is the approach followed by the Garlic [CHS+95] system from IBM Almaden Research Center, the Pegasus [ASD+91] system from HP Labs, and the CoopWARE [Gal99] system from University of Toronto, which is based on the knowledge representation language TELOS [MBJK90].

The *global schema* provides a unified integrated view of all the available information. However, it is usually the case that not all the users need to have access to all the available information. Building a schema that describes all the information that all the users need to access may be infeasible. For that reason, the more flexible architecture of *logical views* has been used [Ull97]. The architecture is depicted in Figure 2.1(b). Above each source, there is a special piece of software, called a *wrapper*, that logically converts the underlying data objects to a common information model. To do that, the wrapper translates queries on the common model into requests in the native query language and schema of the data source. System components, called *mediators*, obtain information

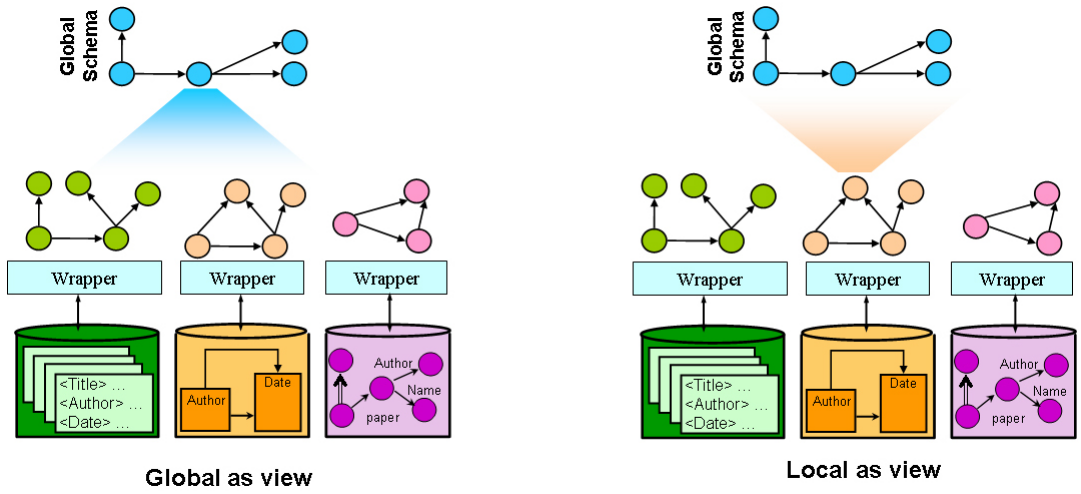


Figure 2.2: GLAV and LAV methods for specifying the global-local schema relationship

from one or more components that may be wrappers or other mediators, and provide it to the components above them that may be users, software applications, or other mediators. A mediator embodies the necessary knowledge for processing a specific type of information. When it receives a query, it is responsible for sending query requests to the appropriate underlying wrappers or mediators, collecting the results, possibly performing some extra processing, and, finally, forwarding them to the user. Data, traditionally, does not exist in the mediator, but a mediator is queried as if the data was stored in it. In a sense, each mediator is a logical view [Ull97] of the data that can be found in the underlying sources. The TSIMMIS system [MHI+95] from Stanford University and the DISCO system [TAB+97] from INRIA use an architecture based on logical views.

A mediator architecture with one level can be seen as a global schema architecture where the schema exported by the mediator is the global schema, and vice-versa. For the rest of the discussion, we will consider these two architectures as one, which we refer to as the *global schema* architecture.

One of the most important aspects of an integration system is how to specify the relationship between the data of the individual sources and their representation as described by the global schema. This specification is usually described through a set of assertions

that are called *mappings* [Len02]. Mappings have the form $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$ where $Q^{\mathcal{S}}$ is a query over the source schema and $Q^{\mathcal{G}}$ a query over the global schema. Intuitively, the assertion $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$ specifies that the result data of query $Q^{\mathcal{S}}$ "corresponds" to the result data of query $Q^{\mathcal{G}}$. Depending on how the term "corresponds" is interpreted, there are different kinds of mappings which will be presented in the following paragraphs.

In real life, data integration systems usually contain more than one data source, in which case the mappings in $\mathcal{M}$ may be between the global schema and any of the local schemas of the sources, or the query $Q^{\mathcal{S}}$ may combine information from multiple local sources [MHI+95, TAB+97].

The goal of a data integration system that follows the virtual approach is, given a query over its global schema $\mathcal{G}$, to translate it to queries over the schemas of the local sources. The mapping constrains this translation. There are two main approaches that have been studied in the literature: the *global-as-view* approach, which is used by Ullman [Ull97], and the *local-as-view* approach which is studied by Levy et al. [LMSS95]. Figure 2.2 shows graphically the main difference between the two approaches. The shaded cone represents the way the views are defined. The top (narrow part) of the cone is a schema defined in terms of the schemas at the bottom (wide part) of the cone.

In the *global-as-view* (GAV) approach, each mapping associates an element of the global schema to a view query over the local source(s). This means that query $Q^{\mathcal{G}}$ in every mapping $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$ is a query that selects only one schema element, e.g., a relational table in the case of a relational schema. The global schema is usually specified implicitly through the interface of the view queries of the mappings. GAV systems favor query answering since a query on the global schema can be translated to queries on the underlying local data sources by substituting the global schema terms used in the query with their associated view. However, such a system is not very flexible to changes. When a new data source is added or an old one removed, redefinition of a number of mappings may be required, and the global schema may have to be modified. Example systems that follow

the GAV approach are TSIMMIS [MHI+95], Garlic [CHS+95] and DISCO [TAB+97].

In the *local-as-view* (LAV) approach, each mapping associates an element of the local schema to a view query over the global schema. This means that the query $Q^{\mathcal{S}}$ in every mapping $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$ is a query that selects only one schema element, e.g., a relational table in the case of a relational schema. The global schema is usually specified by domain or community experts to represent the global knowledge that a user needs to query. The contents of each data source are then described in terms of the global schema through the queries on it. An assumption in LAV systems is that the global schema does not change often, since it describes the domain knowledge. The advantage is that such systems facilitate the addition or removal of local data sources since the only task that needs to be done is to respectively add or remove the mappings that specify the contexts of the data source. However, query answering is much harder in LAV systems, since a query on the global schema needs to be expressed in terms of the queries that use the views of the local sources [LMSS95]. An example system that uses the LAV approach is the Information Manifold [KLSS95] from AT&T Research Labs.

A third approach that has recently appeared and tries to combine the best of both worlds is the so-called *global-local-as-view* (GLAV) [FLM99]. A GLAV mapping is a mapping $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$ in which both queries $Q^{\mathcal{S}}$ and $Q^{\mathcal{G}}$ are complex queries over the source and global schema respectively. Unfortunately, some recent work has shown that composition of GLAV mappings may be undecidable [MH03] for some cases, and that compositions of GLAV mappings, where the queries $Q^{\mathcal{S}}$ and $Q^{\mathcal{G}}$ are conjunctive queries, cannot always be expressed as a GLAV mapping using conjunctive queries [FKPT04]. However, it has been shown that, if constraints are allowed to exist in the global schema, a GLAV mapping may be turned into a GAV mapping and a number of referential constraints in the global schema [CCGL02]. A very recent study has considered mappings where the queries $Q^{\mathcal{S}}$ and $Q^{\mathcal{G}}$ may contain predicates ranging over both the source and global schema [HIST03]. A similar idea has been adopted in the AutoMed project where such

mappings are expressed through a set of schema transformations [MP03]. The term used in AutoMed is *both-as-view* (BAV) mappings [BKL$^+$04].

The majority of the studies found in the literature consider mappings that use only conjunctive queries $Q^{\mathcal{S}}$ and $Q^{\mathcal{G}}$, but more complicated mappings like those that include comparisons and negations [AD98] have also been considered in some cases. Some other systems [VCC00, KLSS95] have also considered Description Logics [BDNS94] from the area of knowledge representation to describe mappings.

Three semantics for the "$\rightsquigarrow$" in a mapping $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{G}}$, have been proposed: *sound*, *complete*, and *exact*. A *sound* mapping is of the form $Q^{\mathcal{S}} \subseteq Q^{\mathcal{G}}$ and means that, if a data value is provided by the query $Q^{\mathcal{S}}$, one can conclude that it should also appear in the global instance. If a value does not appear in the results of query $Q^{\mathcal{S}}$, it does not mean that it does not appear in the global database described by the global schema. Such a mapping is also called *open* [Len02]. A *complete* mapping is one of the form $Q^{\mathcal{S}} \supseteq Q^{\mathcal{G}}$ and means that the data values in the result of the query $Q^{\mathcal{G}}$ on the global database described by the global schema are included in the local source. In other words, if the local source does not contain a specific value, then it is sure that this value does not appear in the global database. Finally, *exact* mappings are mappings that are both sound and complete, i.e., mappings of the form $Q^{\mathcal{S}} = Q^{\mathcal{G}}$. Most of the work found in the literature has considered sound mappings, although complete and exact mappings have also been considered for limited classes of mappings [AD98, GM99].

When a query is expressed on a global schema, the mappings are used to rewrite the query into a query on the source schema(s). Query rewriting has been studied in the context of data integration using logical views (GAV) [Ull97]. Yu and Popa [YP04] have developed a query answering algorithm for GLAV that takes into consideration key and foreign key constraints in XML and relational schemas. The algorithm can handle queries on a virtual target instance if the mappings between the source and the target instance are conjunctive first-order mappings [AHV95].

Building a data integration system is a topic that has attracted a lot of interest in the research literature. Two main topics have received considerable attention: (i) How to build the integrated schema, and (ii) how to specify the mappings between the local and the global schema. A brief overview of the work that has been done in these areas is given in the next two subsections.

2.2 Schema Integration

One of the main issues in building a data integration system is the design of the global schema, that is the schema that will describe the structure of the data that the end user can access in a uniform and transparent way. In the ideal case, this schema will describe exactly the information contained in all the schemas of the local data sources. The trivial solution to this problem is to build a schema that consists of the union of the individual local schemas. The term "union" here is used to describe the schema that results by simply appending the constructs of one schema to another (after a possible renaming to avoid any conflicts). This may result in a poor representation of the data as a whole. A different representation, on the other hand, may result in some loss of information in the integrated data. The problem of designing a "good" integrated schema has been extensively studied.

During the design of the integrated schema, a number of issues need to be resolved:

- **Syntactic Mismatches**: The same piece of information may follow different syntax in different databases. For example, the phone number in one database may contain a dash between the area code and the main phone number, while in another one, it may not.

- **Structural Mismatches:** A piece of information in one source may be represented differently than in another. For example, the name of a person may be a single field in one data source, while in another it may consist of three parts: the last

name, the first name, and the middle initial.

- **Domain or Semantic Mismatches**: The meaning of a value in the same attribute in two different databases may be different. Two databases, for example, may have value 5 in attribute *distance*. However, in the first database it may mean 5 miles while in the second it may mean 5 kilometers.

- **Schematic discrepancies**: Information that is represented as data in one source may be represented as meta-data in another. For instance, in one source, data about people may be stored under one table *Person* that contains an attribute *occupation* for determining the profession of each person. On the other hand, another source may contain one table for each different profession and store in each one of them the people with the corresponding occupation.

- **Dependency Conflicts**: A group of concepts may be related by different dependencies in different databases.

- **Key Conflicts**: Different concepts may be identified by different keys in different sources, and it is not clear which one (if any) should be used when the data from the sources is integrated.

The data administrator must resolve all the above discrepancies and design the global schema in order to shield the end user from them. Batini et al. [BLN86] have recognized that the reason for these discrepancies is mainly the fact that different users may see the same information from different perspectives and develop the databases with different requirements in mind. In addition to this, performance reasons may lead to a fragmented representation of a piece of information. Furthermore, different pieces of information may be stored in different data sources, since for economic reasons, each organization or individual stores in its database only the parts of information that are of particular interest. In this same work, the existing methodologies to attack the above problems

are surveyed. Another interesting survey is the one by Ram and Ramesh [RR99] which argues that the first step in schema integration is to identify the inter-schema relationships, and the next to generate the integrated global schema. The approaches that have been suggested in the literature for the latter can be classified in two categories: *schema restructuring* [MP03] and *view generation* methodologies. The first generates the global schema by restructuring the local schemas until it brings them all into a common consistent form. The disadvantage of this approach is that information from two different sources cannot always be combined. For example, if one data source contains information about the fees of the students and another contains information about the courses that each student takes, it is not easy to create an integrated schema that has for each course, how much money the enrolled students owe, unless the student identifier is included in the schema. The second approach is to define the global schema through a set of view definitions over the local sources. This *view generation* methodology can be applied to the cases where the local sources are heterogeneous and dynamic creation of new data is required.

One important question in the process of generation of the global schema is how much information the schema can describe. The global schema must be such that it will be able to represent all the information of interest from the sources. That is, information is not lost when local data is queried through the global schema, compared to the case where the local schemas are queried directly. On the other hand, the global schema should not model more information than what is in the sources, because that would give the end user the false impression that there is some additional information that can be queried. In order to measure the "relativism" of data structures, i.e., the ability to structure data in different ways, Hull [Hul86] introduced the notion of *information capacity*. Informally speaking, the *information capacity* of a schema describes how much information a schema can describe. For two schemas $\mathcal{S}$ and G, schema $\mathcal{S}$ is said to be *dominated* by G if G has larger information capacity than $\mathcal{S}$. As an example, consider a relational database

schema that consists of a single relation $R(a, b, c)$ and a second one that consists of a single relation $P(a, b, c, d)$. The second schema has at least the information capacity of the first since every instance of the first schema can also be stored in the second and retrieved back from it. Miller et al. [MIR93] have studied the problem of information capacity preservation in schema integration. They used information capacity as a measure of correctness for judging the various methodologies that are used in schema integration. They also showed that there are practical cases in which successful integration can be achieved even if the information capacity is not preserved.

The topic of *schema mapping* we describe in Chapter 4 is fundamentally different than schema integration. In schema integration, the global schema is designed to reflect the semantics of the individual sources. In schema mapping all schemas are given and may describe different data semantics, since they may have been developed independently and with different requirements in mind. The main problem then is to find mappings that specify how the instance data described by the two schemas relate to each other. In schema integration, the mappings are usually explicitly provided, since through these mappings the integrated schema or the sources are described.

2.3 Data Translation

The second major issue in the design of an integration system is the specification of the data translation from the structure described by the local schemas to the structure described by the global schema. Data translation is described by a transformation (or translation) query or program. Note that a mapping is a specification that constrains the translation, while a translation query or program is used to actually perform the translation.

The problem of *data translation* has long been recognized in the research community. One of the first systems to deal with this problem was EXPRESS [SHT+77] developed

by IBM. EXPRESS consists of two languages, DEFINE and CONVERT. The first function works as a DDL (Data Definition Language) and the second as a data translation language. CONVERT has a total of nine operators. Each operator receives as input a data file, performs the respective transformation and generates a new data file. The transformation language of EXPRESS is very limited, but it is recognized as one of the first attempts to deal with this problem. EXPRESS uses a hierarchical data model.

A similar but more recent approach that uses a hyper-graph-based data model is that of the AutoMed [MP03] project. They give a set of predefined changes, i.e., create concept, delete concept, create property, etc. For each such change, there is an associated query that specifies how the data is to be modified to conform to the new schema after the schema change has been applied. Both EXPRESS and AutoMed limit the kind of transformations to those that are predefined.

A more structural approach, introduced by Abu-Hamdeh et al. [AHCM94], uses a structural transformation language called TXL to specify the transformation. TXL was initially designed to describe syntactic software transformations. It works by parsing the data and applying restructuring rules if the parsing is successful.

The problem of describing the transformation becomes harder if the local schema and the global schema are expressed in different data models. Wrappers [TRS97] can be used to ensure that the local and the global schema are all expressed in the same data model. In the absence of wrappers, this issue has to be taken into consideration. The MDM [AT97] system is a tool that enables users to define schemas of different data models and to perform translation of schemas from one model to another. It consists of two main components: the *model manager* and the *schema manager*. A data administrator uses the *model manager* in order to build the appropriate meta-constructs that describe all the data models of interest. For example, there are constructs that represent relations, attributes, classes, keys, foreign keys, etc. The schema manager allows the user to specify schemas over the defined data models, and generate a translation from one to another.

To do that, MDM uses the notion of patterns [AT95]. Intuitively, a pattern can be seen as a graph, and a model is represented as a labeled graph. A pattern can be viewed as a conceptual model description of the schema, i.e., a meta-model. Translations can then be defined as functions that take a pattern and generate a new one. The specific translation language to be used is not critical in MDM, but a good candidate would be a graph-based transformation language. In some later work, Beeri and Milo [BM99] proposed a tree-structured data model for describing schemas, and showed that this is expressive enough to represent relational and XML schemas. The importance of this work is that it recognized the value of the emerging XML syntax [Con98] as a data exchange format.

Assuming that the local and global schemas are in the same model, data translation still remains an interesting issue. Abiteboul et al. [ACM97] provide a formal foundation to facilitate data translation. The advantage of this work is that it provides a way of declaratively specifying the translation. As a common data model, labeled trees are used, since they are expressive enough to represent relational and object-oriented models. Relationships between the two schemas are expressed through *correspondence rules*. A correspondence rule consists of two parts: a head and a body. The body specifies how parts of the schema should be selected (by specifying a subtree of the tree representation of the schema) and the head specifies what will be generated. Both the body and the head may contain two special operators: concatenation and merging. Concatenation is used to put the values of two different elements together, while merging is used to specify that two nodes of the tree model represent the same real world entity and should be merged into one. The following rule, for instance, specifies the selection of a person and courses that this person is taking, and generates an entry that has the name of the person (which consists of the concatenation of the first and last name) and also the courses that the person is taking. It is defined on a tree-based model where the variables $\&X_i$ represent the nodes and the names represent labels on the edges. For example, $\&X_A$ *name* $\&X_B$ models two nodes $\&X_A$ and $\&X_B$ with an edge between them named "name". The head

of the rule specifies how the results will be formed, and the body specifies from what parts of the data these results will be derived. The *is* constructor specifies that the values $\&X_2$ and $\&X_7$ represent the same real world entity, the student number.

$\&X_{15}$ *person* $\{\&X_5$ *name* $\&X_{12}$, $\&X_{13}$ *courses* $\&X_3\} \leftarrow$
$\&X_0$ *course* $\&X_1$ *studentNo* $\&X_2$, $\&X_3$ *courseName* $\&X_4$,
$\&X_5$ *student* $\{\&X_6$ *studentNo* $\&X_7$, $\&X_8$ *fname* $\&X_9$, $\&X_{10}$ *lname* $\&X_{11}\}$,
concat($\&X_{12}$,$\&X_9$,$\&X_{10}$), is($\&X_2$,$\&X_7$)

Abiteboul et al. also show that the problem of generating an extension of a schema (i.e., an instance of the global schema) given a set of correspondence rules (i.e., mappings) and a local instance (an instance of a local database) is in the general case undecidable, although tractability can be achieved under certain restrictions [ACM97]. Their data model and its translation rules have been used in the TranScm system [MZ98]. TranScm is a system built to simplify heterogeneous translation. It was one of the first systems to consider the use of schema information to help automate data translation, and is perhaps the closest to our approach presented in Chapter 4. TranScm identifies matches between two schemas and derives the translation functions. The approach is based on a set of predefined matching rules that describe the most commonly used transformations. The rules are mostly structural, ignoring constraints. In addition, the expressive power of the rules is limited in that they can detect only local structural differences. More advanced mappings that include joining disparate portions of the schema cannot be created (and indeed were not the focus of TranScm). In contrast, the approach that will be presented in Chapter 4 can generate complex join queries with nesting.

Most of the research on data translation has focused on the development of expressive translation languages to specify the mappings between the local and the global schema, rather than on automating the creation of programs or queries expressed in these languages. The data translation languages, including rule-based correspondence languages,

have powerful pattern matching primitives that can facilitate data restructuring and translation. The conversion language used in the YAT system [CDSS98] has also adopted a tree-like data model. It is strongly typed, allowing static type checking, and it has the advantage of reordering collections and detecting cycles in the transformation rules. The WOL language [DK97] was the first to take into consideration integrity constraints in the language design. Through the interaction between the WOL transformation rules and constraints, one can achieve information preserving transformations. This proves useful in integration when one needs to impose some constraints on the transformed data, e.g., specify that an attribute is a key. However, these constraints may not be part of the schema, either because of incomplete schema design, or because these constraints were not known at the time of schema design, or because it is not possible to express such constraints on the schema. A similar approach is followed by the AIG translation language for XML data [BCF+03]. AIG considers constraints that may exist in the global schema, however, it does not consider the constraints during the mapping construction, but only during the evaluation of the mappings over the local sources. Furthermore, the mappings need to be defined explicitly by the user. No automated mapping construction process is provided. Query languages can also be used as translation languages, e.g., XQuery [CCDF+01] or XSLT [Cla99].

In a number of cases, getting the data from the local sources and reformulating it to bring it into the structure described by the global schema may not be enough. New values may need to be generated. A method for generating new values is to use Skolem functions like those introduced in ILOG [HY90]. A *Skolem function* is an injective function and can be used to produce unique identifiers. For example, the function $Sk(name)$ generates a different value for each different name. Skolem functions will be used extensively in Chapter 4 for generating values for schema elements that are required to have values, but their value is not specified by the mapping. In this thesis, we use nested relational model queries to express the data transformations. The expressive power of these queries

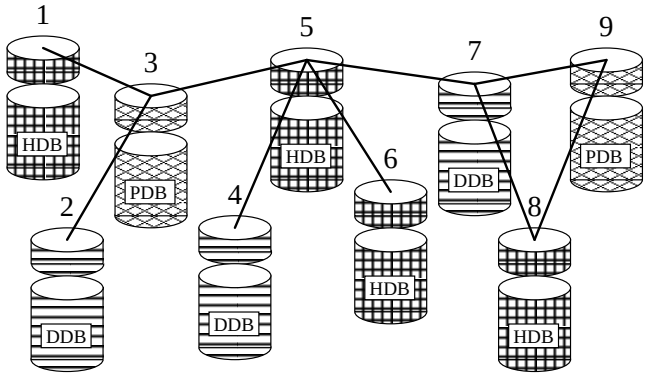


Figure 2.3: Architecture of a peer-to-peer system.

allow us to describe a large portion of the transformations that are commonly used in information integration scenarios.

2.4 Data Exchange

The advent of the Web has introduced new applications for data requirements. More and more organizations and individuals have started publishing their data on the Web, making it an enormous distributed data repository. Existing integration systems like those presented in the previous sections do not perform well on that scale. New systems were introduced like *peer-to-peer* systems [TIM+03, KAM03, AKK+03, SMLN+, LRS02]. Also, new exchange format standards were specified [DD99]. In peer-to-peer systems, each data source, referred to as a *peer*, is willing to share its data with other peers. Its data model, format or schema may be different from those of other peers with which it communicates. To achieve efficient data exchange, data needs to be translated from the format of one peer to the format of another. This can be done through a set of mappings. In standardized communities, data needs to be exchanged over the network in a predefined format. Applications that need to communicate data have to translate it from its native format to the predefined one and vice-versa.

The above applications, along with many others, led researchers to start studying

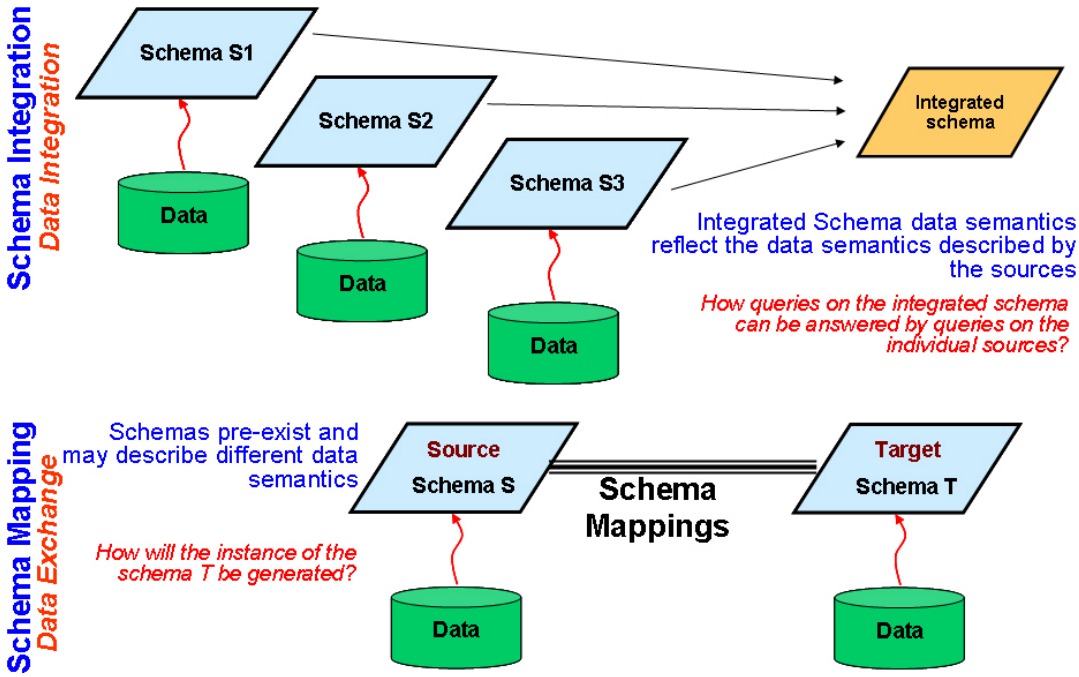


Figure 2.4: Difference between data integration and data exchange.

the problem of *data exchange* [FKMP03b]. In *data exchange*, data structured under one schema (which will be referred to as the *source schema*), must be restructured to conform to the structure and the constraints described by a second schema (which will be referred to as the *target schema*). Mappings need to be specified between the source and the target schema in order to perform this translation. A data exchange system can be seen as an integration system where the source schema plays the role of the local data source, and the target schema plays the role of the global schema. Despite the many similarities between integration and data exchange systems, the data exchange problem is fundamentally different from data integration. In data integration, the goal is to provide a uniform interface to multiple autonomous data sources. Based on this, most of the work in data integration has concentrated on query rewriting, that is, given a query on the integrated schema, specify how this query is to be translated to a set of queries on the individual sources. On the other hand, the *data exchange problem* is defined as the problem of determining an instance of the target schema given an instance of a source

schema and a schema mapping, i.e., the target instance is materialized. The source and target schema, the instance of the source schema, and a set of mappings form a *data exchange system*. The problem has been formally defined by Fagin et al. [FKMP03b] who have considered data exchange when there are constraints on the target schema. The constraints on the target schema restrict the data that can populate it. With the exception of Cali et al. [CCGL02] and Benedikt et al. [BCF$^+$03], the majority of the work in data integration and data exchange has ignored the existence of constraints in the global or the target schema. The mappings considered by Fagin et al. are *source-to-target dependencies* of the form $\forall Q^{\mathcal{S}} \rightarrow Q^{\mathcal{G}}$ where $Q^{\mathcal{S}}$ and $Q^{\mathcal{G}}$ are conjunctive queries over the source and target schema, respectively. The constraints that are considered in the target schema can be either tuple or equality generating dependencies [AHV95]. In the general case, and for a given data exchange system, there may be more than one instance of the target schema that satisfies the mappings and the constraints of the target schema. These mappings are referred to as *solutions*. The minimal instances in terms of generating the least redundant instance in the target schema are characterized by Fagin et al. as *universal solutions*. An extension of this work [FKP03] characterizes the "best" among the universal solutions as the *core*, and has been inspired by the mapping generation tool Clio [MHH00, PVM$^+$02b], in which the mapping generation and data translation framework of Chapter 4 have been implemented.

An area that has been investigated within the context of universal solutions and the core is query answering. In particular, suppose that q is a query over the target schema $\mathcal{T}$, and $\mathcal{I}$ is an instance over the source schema $\mathcal{S}$. What does answering q with respect to $\mathcal{I}$ mean? In the case of a data integration system, this is related to the notion of *certain answers*. That is, a query on a global schema $\mathcal{T}$ is rewritten into a query on schema $\mathcal{S}$ that, when executed on $\mathcal{I}$ returns the values that appear in all the valid instances of schema $\mathcal{T}$ that satisfy the mappings between the local and the global schemas. The notion of certain answers can be extended in the case of data exchange. Clearly, there

may be many ways that a target schema can be materialized, given a source instance $\mathcal{I}$ and a mapping, i.e., there may be many solutions. What is not clear is what the answer of a query on the second schema will be (since it will depend on the particular instance that was chosen to be materialized). Fagin et al. consider as an answer of a query q over the schema $\mathcal{T}$ only the values that appear in every answer set of query q in all the solutions. This is similar to the notion of certain answers in data integration but in the context of data exchange. In our work, we do not deal with query answering, However, we provide the framework from that. Chapter 4 describes how mappings can be constructed in order to build a data exchange system. It also provides a method for creating a target instance. That target instance can provide one semantics for query answering.

2.5 Schema Evolution

A common assumption in the work on data integration is that the schemas of the databases and, consequently, the mappings remain relatively static. This is why most of the data integration work has concentrated on query answering. In the current Web environment where there is no centralized authority, data sources may independently, and even frequently, change their contents and their schema. For example, tuples may be added or deleted in the data sources, and relations and schema elements may be added, deleted, or moved in the schemas. Data changes may affect the materialized view data or a materialized target schema instance in a data exchange system. Schema changes may also affect materialized views or target schema instances, but also view definitions and mappings.

A lot of work on schema evolution has been done in the area of *object-oriented database management systems (OODBMS)*. Object-oriented databases need to allow a wide variety of changes to their schema and modify the instance data automatically to conform to the new version of the schema. One of the main issues is to minimize the cost of updating

the instance data after a schema change. Banerjee et al. [BKKK87] gave a taxonomy of the changes that may occur in OODBMS, and provided an implementation for each one of them. The changes supported were local changes to the classes of the object-oriented schema or their extension, such as renaming of a method, changing the class of an object, changing class hierarchy, or changing the code of a method. For each change, the authors gave a description of how the instance data should be updated. They also provided a set of rules that guarantee the consistency of the database. Proving the completeness of the set of changes described in this work is claimed to be neither feasible nor meaningful, since it will always depend on the focus of interest. For example, in the work of Banerjee et al., the focus is mostly on the manipulation of the class lattice or the structure of an object. Lerner [Ler00] extended the above work to include complex changes that span multiple classes and provided templates for the most common changes. Examples of changes that span more than one class are moving an attribute from one type to another, or splitting a class into two and vice-versa. Lerner exploited the power of object-oriented languages to describe a template for each kind of change. The template describes exactly how the instance data is to be updated. When a schema change occurs, the corresponding template is activated and executed. Despite the expressive power of Lerner's model, the approach is limited to only the changes for which a template has been defined.

A *warehouse* is a repository that implements a materialized view. Materialized views must be maintained as the data of the base tables on which the views are defined is modified. Materialized views and data warehouses are becoming increasingly important in practice. In order to satisfy different performance and data requirements, various view maintenance policies have been proposed in the literature [CW91], and many research issues have been studied [Wid95, RAJB+00, GM95]. Quass et al. [QGMW96] exploited key and foreign key information to maintain select-project-join views with respect to insertions, deletions and updates. By using some auxiliary views, they were able to identify certain classes of views that can be self-maintainable. Some of the authors have also stud-

ied update anomalies that may occur in view maintenance [ZGMHW95]. For example, the insertion of a tuple may require more data to be updated than what is in the tuple. The authors developed a mechanism to postpone a change until the required extra information becomes available. More complicated views require more complicated techniques to be maintained. Palpanas et al. [PSCP02] developed an algorithm for maintaining views that contain aggregate functions. Other work has considered the use of auxiliary data structures [BLT86], production rules [CW91], or selective recomputation [MQM97].

View adaptation [GMR95, MD96] is a variant of *view maintenance* that investigates methods of keeping the data in a materialized view up-to-date in response to changes in the view definition itself. The goal is to find ways to do this other than by recomputing the view from the base relations. One way to attack this problem is to keep a small amount of additional information beyond the data in the view, such as keys of participating relations [GMR95] or join counts [MD96]. View adaptation is a part of a broader problem called *view management* that includes all the issues related to the creation and manipulation of views, e.g., reusing views to optimize query answering or data storage in cases of materialization [KR99]. View management is a long standing problem in the commercial database system community where there is a need for detecting views (materialized or not) that become invalid due to schema changes in the base tables [BHL83].

A different approach to schema evolution has been followed by McBrien and Poulovassilis [MP02] who combine schema evolution and schema integration in one unified framework. By using a series of primitive schema transformations, one can map a local schema to a global schema. Each transformation must be accompanied by a query that describes its semantics. This query has to be manually specified by the user. Their approach enables easy composition of the transformations and permits optimization. In our approach, the user does not need to manually specify such queries.

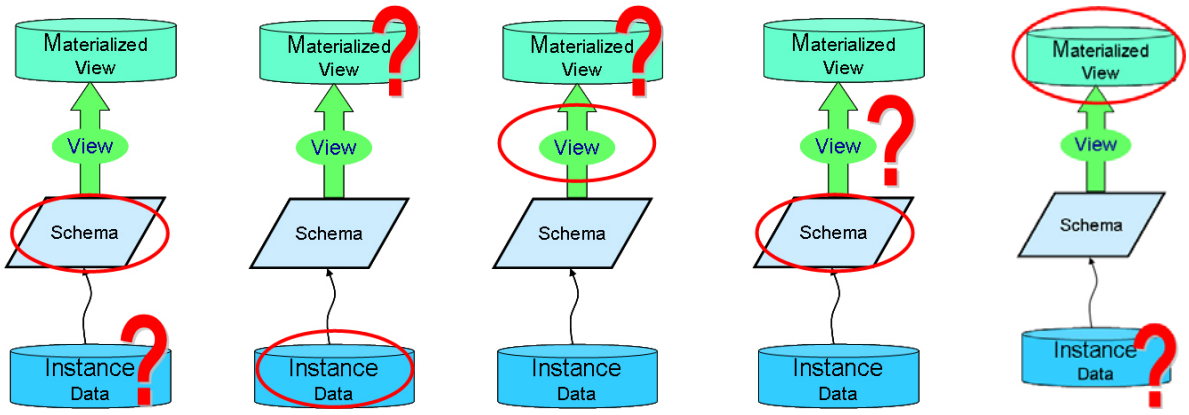
In the EVE system [LNR02], the *view synchronization* problem was addressed, that

is how a view definition has to be updated when the base relational schema is modified. When a view is initially defined, the data administrators characterize the parts of the view definition as *can-be-deleted*, *mandatory*, etc. For example, if a view uses an attribute of a relation in its select clause that has been characterized as *mandatory*, and that attribute is deleted, then the view should also be deleted. On the other hand, if it has been characterized as *can-be-deleted*, then the attribute is removed from the view definition. In many cases, inclusion dependencies may be used to replace attributes that have been deleted with others. In EVE, a user who defines a view is required to specify how the system should behave under changes. Furthermore, the changes supported are restricted to only deletion and renaming. Changes such as moving and copying attributes as well as constraint changes are not considered. Furthermore, the EVE system is restricted only to views and not complicated mappings like the GLAV mappings that are considered in Chapter 5.

Apart from view synchronization, all the other approaches that were presented in this section deal with data, that is, they try to preserve data consistency in reaction to some change or evolution that takes place. On the other hand, the mapping adaptation problem that we define and the view synchronization problem deal with changes at the schema level. The goal is to maintain the meta-data (view or mapping definition) when a change occurs to the schema. Figure 2.5 provides a graphical description of how the problems presented in this section differ. The circle indicates where the change takes place, and the question mark specifies what is updated or maintained as a result of the change that took place.

2.6 Meta-data Management

Representing and querying meta-data is an old problem. Almost 25 years ago, Codd recognized the need to capture and query the meaning of the data in a more formal



schema evolution | view maintenance | view adaptation | mapping adaptation or view synchronization | view update

Figure 2.5: Differences of the research areas that deal with data or schema evolution.

way [Cod79]. In the last few years, a number of proposals have appeared in the literature that elevate various forms of meta-data to first-class citizens of the query language. Hachem et al. [HQGW93] propose a system for modeling data derivation processes, but do not associate them with the actual data. Mihaila et al. [MRT98] annotate the data with quality parameters such as accuracy, frequency of updates, etc. Queries containing quality specifications can then be executed on that data. A similar approach is followed by Chawathe et al. [CAW98], who explicitly represent database changes, thereby allowing queries to be executed on different versions of database instances over time. In a similar fashion, the work presented in Chapter 6 explicitly models schemas and mappings, and thus permits queries to trace the origin of the transformations used to derive the data. Data annotations have also been used by Wang and Madnick [WM90] to keep track of all the data sources from which a specific data value originates. This approach keeps information only about the data sources and cannot detect whether two values from the same source originate from the same element of that source, or whether they went through the same transformation. Querying data and meta-data (schemas) has also been

proposed by Lakshmanan et al. [LSS96], who consider schemas as first class citizens of a query language but the main scope of this work was schema restructuring and not finding the origin of data.

Cui and Widom [CW00] define *data lineage* as the tuples in the base tables of a view that justify the existence of a specific value in the view. Buneman et al. [BKT01] refer to data lineage as *why-provenance* in order to distinguish it from the *where-provenance*, which is defined as the exact part of the why-provenance from which a view value was extracted. For example, where-provenance can be a set of facts from an attribute of a tuple, or an element of an XML tree. One the other hand, why-provenance is the set of complex structures (i.e., whole tuples or proof trees [AHV95]) that are responsible for the appearance of a data element in the view. Not all the parts of such a structure justify the existence of a value in a view. In Chapter 6, we define a new form of provenance that captures exactly this notion. Specifically, we define *what-provenance* as only the parts of the where-provenance that are needed to justify the existence of the value. Furthermore, data provenance is data-oriented in the sense that it considers the provenance of a view value as a set of data values in the base schema. However, in many applications, it is enough to know the parts of the base schema from which the view value originates, but not the originating values themselves. In Chapter 6, we capture the information about the schema elements from where the values originate, instead of the actual values. This relaxation greatly reduces the complexity of the problem since it does not involve the costly operation of query inversion required in finding the data provenance. Finally, apart from the originating data values that data provenance considers, we also consider the transformations used to generate the value of interest in the integration. This schema-level and transformation provenance does not replace, but rather complements, data provenance. The schema level provenance is the first step for a user to realize the origin of the data (schema elements and transformation). Then, if the actual originating values are needed, data provenance techniques can be used.

Currently [BCTV04], there are emerging efforts to build annotation systems (e.g., Annotea [KKPS01]), in which data annotations, defined by users, propagate along with the data, in order to facilitate the use of superimposed information [BDM02]. The problem of propagating annotations from a view back to the source has been studied by Buneman et al. [BKT02]. In Chapter 6, annotations are queried as regular data so the complexity of the problem there is not more than the complexity of evaluating a query over an instance.

Model management [Ber03] is a research area in which meta-data are managed as first-class citizens [MRB03] which means that a number of primitive algebraic operations, such as match [RB01], compose [MH03, FKPT04] or merge [RP03], are defined on them.

Schema matching is the usual first step in a mapping generation process. In schema matching, two schemas are compared and the result is a set of binary relationships between their elements. To perform such a matching, different techniques have been proposed in the literature. Rahm and Bernstein provided a nice survey of the most common such techniques [RB01]. Melnik et al. [MGMR02] suggest a structural algorithm that can be used for matching schemas. The algorithm is based on the following idea. First, the schemas to be matched are converted into directed labeled graphs. These graphs are used in an iterative fix-point computation whose results indicate what nodes in one graph are similar to nodes in the second graph. For computing the similarities, they rely on the intuition that elements of two distinct models are similar when their adjacent elements are similar. In other words, a part of the similarity of two elements propagates to their respective neighbors. The spreading of similarities in the matched models is reminiscent to the way Internet Protocol packets flood the network in broadcast communication. For this reason, Melnik et al. call their algorithm *similarity flooding*. With such a tool, matching is not done entirely automatically. Instead, the tool assists human developers in matching by suggesting plausible match candidates for the elements of a schema. Using a graphical interface, the user adjusts the proposed match result by

removing or adding lines connecting the elements of two schemas.

The Cupid [MBR01] and COMA [DR02] systems use a number of different approaches and combine their output in order to achieve better matching suggestion. They employ algorithms that use linguistic reasoning to match attributes based on their names or their place in a hierarchical structure. In addition, Cupid uses information about types, optionality, cardinalities, etc. It also tries to match schema constraints such as keys or referential constraints. He and Chang [HC03] noticed that, in specific communities, the schema vocabulary tends to converge at a relatively small size. For such cases, they propose the use of probabilistic techniques to infer matchings.

If some data is already stored in the target schema, one may also use a data mining technique that matches data values or their characteristics to find binary relationships between the schema elements [NHT+02]. The LSD system [DDH01] uses a multi-level learning scheme to find 1:1 binary relationships between XML DTD tags. A number of base learners that use different instance-level matching schemes are trained to assign tags of a mediated XML schema to data instances of a source schema. Then, a meta-learner combines the predictions of the base learners. Techniques that are data-value based are important for cases where determining correspondences based on the names of the schema elements is hard. Based on that observation, Kang and Naughton [KN03] measure the pair-wise attribute correlations in the schema elements to be matched and construct a dependency graph as a measure of the dependency between attributes. Then they find matching node pairs in the dependency graphs by running a graph matching algorithm.

Although schema matching is aided tremendously by the many recent and interesting results in the research community, full automation of this task has not been achieved. The reason is that even matches as plausible as `Company` to `company` can be deemed as incorrect by a data warehouse designer who knows that the first schema element is used to represent names of companies (corporations) while the second to represent the names

of close friends of a person. In such cases, the mappings suggested by a schema matching tool may be incorrect or incomplete. The tool however, may provide some hints to warn the user about potentially incorrect matchings.

The work of this thesis can be seen as a contribution to a broader framework of model management. Chapter 4 presents how mappings can be generated. Chapter 5 presents a framework that can be viewed as an update operation on mappings. Finally, Chapter 6 considers schemas and mappings as first-class citizens of the repository and the query language. This allows schemas and mappings to be queried as regular data.

Chapter 3

The Problem and the Common Model

Generating and managing mappings is a laborious, time-consuming, and error prone task. This chapter demonstrates, through a specific example, how complex a mapping can become, illustrating the significant amount of effort required on the part of data administrators to perform mapping generation and maintenance. Furthermore, Sections 3.2 to 3.4 present the data model that will be used in this work as a common platform to express schemas and mappings from heterogeneous sources.

3.1 A Schema Mapping Example

Consider an XML data source that contains information about projects, grants and companies. The structure of its data is described by the schema `S` that appears on the left of Figure 3.1. For convenience, the schema is presented in a nested relational representation that will be described in details in Section 3.2. In short, a nested relational schema consists of a set of records, choices and set types. A record describes a structure that has a finite number of sub-structures, a choice describes a structure that has only one substructure out of a finite number of substructures, and a set describes a collection of

homogeneous structures. The XML Schema of schema S is given in Figure 3.2. Figure 3.3 describes an instance of that schema. The close tags have been omitted due to space restrictions. As can be seen, the data source contains a set of projects. Each project (`project`) has a name (`name`) and the year (`year`) that the project started. The database also contains a set of grants. For each grant (`grant`), it includes the grant identifier (`gid`), the principal investigator (`pi`), the budget (`budget`), and the recipient (`recipient`) of the grant. Each grant also has a sponsor. Each sponsor (`sponsor`) can be either a private (`private`) individual or a government organization (`government`). The values of the private and government sponsor elements are actually contact identifiers. A contact identifier (`cid`) uniquely identifies some contact information (`contact`). Contact information includes an email address (`email`) and a phone number (`phone`) for each contact. This is actually specified in the XML Schema by the key definition *contactId* and the keyref specification *f2* and *f3*. In Figure 3.1, this information is indicated by the curved continuous arrows. Foreign keys, or in general referential constraints, are considered to be part of the schema. They can be found by looking at the schema definition, i.e., by querying the catalog tables of a relational schema or by referring to its Data Definition Language (DDL) statements. Alternatively, foreign keys and keys may be specified by a user or discovered using a dependency miner [KMRS92]. In addition to information about projects and grants, the data source contains information about companies. For each company (`company`), it records the company identifier (`coid`) that uniquely identifies a company, and the company name (`cname`), as well as the CEO (`CEO`) and owner (`owner`). The CEO and owner are actually referencing some information about people by referring to a social security number (`SSN`) which uniquely identifies a person (`person`). For each person, the name (`name`) and, optionally, the address (`address`) are also recorded.

Consider the case where we would like to use the data of our source database to populate a second one. Let the schema of that second database be the one that appears on the right hand side of Figure 3.1 (the schema `T`). Notice that schema `T` describes

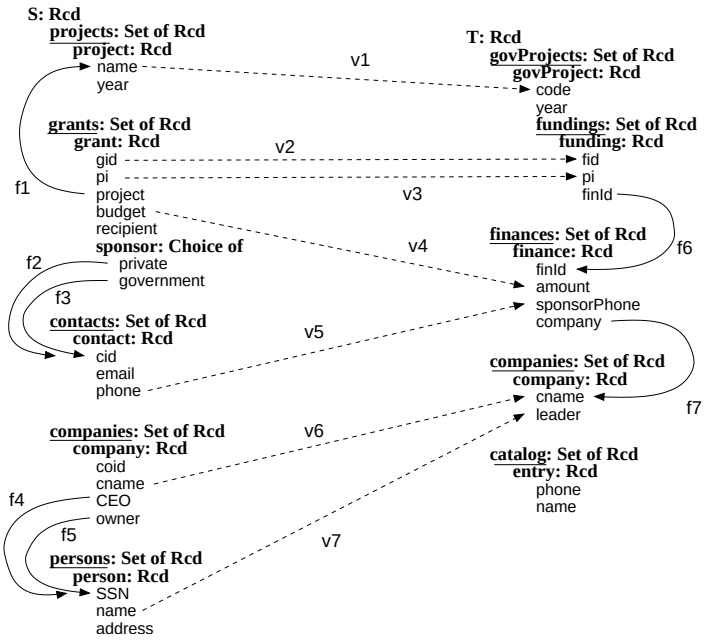


Figure 3.1: A schema mapping example

more or less the same information as our initial source schema, but it is structured differently. In particular, schema `T` records only projects sponsored by a government organization (`govProjects`). Each such project (`govProject`) has a code (`code`), a year (`year`) and a set with the funding amounts that this project has received. Each such funding (`funding`) has a funding identifier (`fid`), a principal investigator (`pi`), and a value (`finId`) that references the identifier of some financial information (`finances`). In a finance entry, the dollar amount (`amount`) of the funding, and the phone number of the sponsor (`sponsorPhone`) are recorded. It also contains a value (`company`) that refers to the name of the company (`cname`) related to this funding. Each company records, apart from its name, the name of its leader (`leader`). Finally, the database contains name (`name`) and phone (`phone`) number information through a set of catalog entries (`entry`).

The dotted arrows between the two schemas in Figure 3.1 indicate how the two schemas correspond. The arrow v_1, for example, indicates that the project code in the second database corresponds to the project name in the first. Correspondence will be formally defined in the next chapter, but informally speaking, "corresponds" means that

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="S"> <xs:complexType>
      <xs:sequence>
         <xs:element name="project" minOccurs="0" maxOccurs="unbounded"> <xs:complexType>
            <xs:all>
               <xs:element name="name" type="xs:string"/>
               <xs:element name="year" type="xs:string"/>
         <xs:element name="grant" minOccurs="0" maxOccurs="unbounded"> <xs:complexType>
            <xs:sequence>
               <xs:element name="gid" type="xs:string"/>
               <xs:element name="pi" type="xs:string"/>
               <xs:element name="project" type="xs:string"/>
               <xs:element name="budget" type="xs:string"/>
               <xs:element name="recipient" type="xs:string"/>
               <xs:element name="sponsor"> <xs:complexType>
                  <xs:choice>
                     <xs:element name="private" type="xs:string"/>
                     <xs:element name="government" type="xs:string"/>
         <xs:element name="contact" minOccurs="0" maxOccurs="unbounded"> <xs:complexType>
            <xs:all>
               <xs:element name="cid" type="xs:string"/>
               <xs:element name="email" type="xs:string"/>
               <xs:element name="phone" type="xs:string"/>
         <xs:element name="company" minOccurs="0" maxOccurs="unbounded"> <xs:complexType>
            <xs:all>
               <xs:element name="coid" type="xs:string"/>
               <xs:element name="cname" type="xs:string"/>
               <xs:element name="CEO" type="xs:string"/>
               <xs:element name="owner" type="xs:string"/>
         <xs:element name="person" minOccurs="0" maxOccurs="unbounded"> <xs:complexType>
            <xs:all>
               <xs:element name="SSN" type="xs:string"/>
               <xs:element name="name" type="xs:string"/>
               <xs:element name="address" minOccurs="0" type="xs:string"/>
   <xs:key name="projectId"> <xs:selector xpath="./project"/> <xs:field xpath="name"/>
   <xs:key name="grantId"> <xs:selector xpath="./grant"/> <xs:field xpath="gid"/>
   <xs:key name="contactId"> <xs:selector xpath="./contact"/> <xs:field xpath="cid"/>
   <xs:key name="companyId"> <xs:selector xpath="./company"/> <xs:field xpath="coid"/>
   <xs:key name="personId"> <xs:selector xpath="./person"/> <xs:field xpath="SSN"/>
   <xs:keyref name="f1" refer="projectId">
      <xs:selector xpath="./grant"/>
      <xs:field xpath="project"/>
   <xs:keyref name="f2" refer="contactId">
      <xs:selector xpath="./grant/sponsor"/>
      <xs:field xpath="private"/>
   <xs:keyref name="f3" refer="contactId">
      <xs:selector xpath="./grant/sponsor"/>
      <xs:field xpath="government"/>
   <xs:keyref name="f4" refer="personId">
      <xs:selector xpath="./company"/>
      <xs:field xpath="CEO"/>
   <xs:keyref name="f5" refer="personId">
      <xs:selector xpath="./company"/>
      <xs:field xpath="owner"/>
```

Figure 3.2: An XML Schema for the source schema S.

```
<?xml version="1.0" encoding="UTF-8"?>
<S xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                            xsi:noNamespaceSchemaLocation="S.xsd">
   <project> <name> DB2
             <year> 1996
                                              <contact> <cid>   NRC
   <project> <name> SQL Server                          <email> info@nrc-cnrc.gc.ca
             <year> 1998                                <phone> 613-993-9101

   <project> <name> Oracle                  <contact> <cid>   Bell Canada
             <year> 1989                              <email> bcecomms@bce.ca
                                                      <phone> 800-668-6878
   <grant>   <gid>        g1
             <pi>         Bernstein         <contact> <cid>   24680
             <project>    SQL Server                <email> parmisario@us.ibm.com
             <budget>     10K                       <phone> 888-839-9289
             <recipient>  Microsoft
             <sponsor>                      <contact> <cid>   NASA
                <government> NSF                    <email> pub-inc@hq.nasa.gov
                                                    <phone> 650-604-6442
   <grant>   <gid>        g2
             <pi>         Pirahesh          <company> <coid>   IBM Yorktown
             <project>    DB2                       <cname>  IBM
             <budget>     8K                        <CEO>    24680
             <recipient>  IBM                       <owner>  987654
             <sponsor>
                <government> NRC            <company> <coid>   IBM Almaden
                                                    <cname>  IBM
   <grant>   <gid>        g3                       <CEO>    24680
             <pi>         Zuzarte                  <owner>  987654
             <project>    DB2
             <budget>     15K               <company> <coid>   MS
             <recipient>  IBM                       <cname>  Microsoft Corp
             <sponsor>                              <CEO>    13579
                <government> NASA                   <owner>  123456

   <grant>   <gid>        g4                <person>  <SSN>   123456
             <pi>         Sousa                     <name>  Gates
             <project>    SQL Server
             <budget>     5K                <person>  <SSN>   13579
             <recipient>  Microsoft                 <name>  Ballmer
             <sponsor>
                <private> BUL               <person>  <SSN>   24680
                                                    <name>  Palmisano
   <contact> <cid>   NSF
             <email> info@nsf.gov           <person>  <SSN>   987654
             <phone> 703-292-5111                   <name>  Lineen
```

Figure 3.3: An XML instance of the source schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<T xmlns:xsi="http:..."
   xsi:noNamespaceSchemaLocation="T.xsd">

   <govProject> <code>    DB2
                <funding> <fid>     g2
                          <pi>      Pirahesh
                          <finId>   ?????
                <funding> <fid>     g3
                          <pi>      Zuzarte
                          <finId>   ?????

   <govProject> <code>    SQL Server
                <funding> <fid>     g1
                          <pi>      Bernstein
                          <finId>   ?????

   <finance>    <finId>        ?????
                <amount>       15K
                <sponsorPhone> 650-604-6442
                <company>      ???

   <finance>    <finId>        ?????
                <amount>       8K
                <sponsorPhone> 613-993-9101
                <company>      ???

   <finance>    <finId>        ?????
                <amount>       10K
                <sponsorPhone> 703-292-5111
                <company>      ???


   <company>    <cname>   Microsoft Corp.
                <leader>  Gates

   <company>    <cname>   IBM
                <leader>  Lineen
```

(a) Without invention of new values

```
<?xml version="1.0" encoding="UTF-8"?>
<T xmlns:xsi="http..."
   xsi:noNamespaceSchemaLocation="T.xsd">

   <govProject> <code>    DB2
                <funding> <fid>     g2
                          <pi>      Pirahesh
                          <finId>   NV2
                <funding> <fid>     g3
                          <pi>      Zuzarte
                          <finId>   NV1

   <govProject> <code>    SQL Server
                <funding> <fid>     g1
                          <pi>      Bernstein
                          <finId>   NV3

   <finance>    <finId>        NV1
                <amount>       15K
                <sponsorPhone> 650-604-6442
                <company>      NV4

   <finance>    <finId>        NV2
                <amount>       8K
                <sponsorPhone> 613-993-9101
                <company>      NV5

   <finance>    <finId>        NV3
                <amount>       10K
                <sponsorPhone> 703-292-5111
                <company>      NV6


   <company>    <cname>   Microsoft Corp.
                <leader>  Gates

   <company>    <cname>   IBM
                <leader>  Lineen

   <company>    <cname>   NV4
                <leader>  NV7

   <company>    <cname>   NV5
                <leader>  NV8

   <company>    <cname>   NV6
                <leader>  NV9
```

(b) With invention of new values

Figure 3.4: An XML instance of the target schema.

these two elements describe the same real world concept, in this particular case the name of a project. This example indicates that, due to the heterogeneity and the different requirements for which two database schemas may have been developed, the same real world entity may be represented in very different ways in the two databases. Contrapositively, information that appears to be the same, may not really be. For instance, in both schemas, projects have an element `year`, but the lack of an arrow between these two elements in Figure 3.1 indicates that they do not represent the same concept. The `year` in the first schema may represent the year that the project started, while in the second it may represent the year that the project was approved. Another way one can read such a correspondence is "whatever is called project `name` in the first schema, is called project `code` in the second". Thinking in terms of populating the second schema using the data from the first, the dotted arrow can be interpreted as stating that the project `code` in the second schema should be populated with project `name` values from the first schema.

The problem is to find the mappings between the source and the target schema that satisfy the correspondences. It will be shown in the next chapter that there may be many such mappings. Determining which one is the "right" one is a real challenge and it is one of the issues we will study in Chapter 4.

In a data exchange setting, the mappings are used to generate an instance of the target schema by populating it with data from the source schema. The generated instance of the target schema must be such that the mappings are satisfied. For our running example, one such instance of the target schema T is illustrated in Figure 3.4, where the close tags have been omitted. Since the two schemas may have been independently developed, they may not describe exactly the same information. For example, it can be noticed that the source schema contains information about `email`, while the target schema does not. Similarly, there are some elements in the target schema for which there is no corresponding element in the source schema. For instance, there is no `finId` element in the source schema. However, the specifications of the target schema indicate

that element `finId` is mandatory and cannot be set to the null value. Thus, a value should be assigned to the `finId` element when the target schema is populated with the data from the source. In the instance of Figure 3.4(a), the values of such elements have been marked with question marks. Since their value cannot be specified from the source instance, i.e., they contain unknown data, it is natural to assume that new values need to be invented to model this incomplete data. These new values should satisfy certain conditions. For example, the specifications of the target schema indicate that the value of element `finId` under `funding` is a reference to a funding identifier. This means that its value must be one of the funding identifiers that the target instance will contain after it has been populated. Furthermore, notice that the grant `g1` is provided (sponsored) by NSF, which has the contact phone number `703-292-5111`. When the target schema is populated, it is desirable for the association between grant `g1` and the phone number `703-292-5111` to be preserved. This means that the new value that will be invented for element `finId` of the `funding` with `fid` value `g1` should not be just any financial identifier, but the one that contains the phone number `703-292-5111`. Furthermore, each occurrence of financial information is related to some company through the value of the element `company` under `finances`. Unfortunately, the source schema does not provide any information to specify which company is related to a specific value of `finances`. With all the above in mind, a possible set of new values that can be generated to fill in the elements with the question-marks is indicated in Figure 3.4(b). In Chapter 4, we will present an algorithm that generates such values by using *Skolem functions* [HY90].

Another question that is not straight-forward to answer from the above example, is which projects should populate the target schema. Looking at the element names, one can guess that the target schema should be populated with projects funded by the government. A question that arises is what happens with the projects that have no funding. For instance, for the project `Oracle` in Figure 3.3 there is no grant. Should it be included in the target schema? Rephrasing the same question, do we want to populate

the target schema with every project, and for each such project, nest within it the set of all government fundings it may have, or should we consider only projects that have some government funding. Furthermore, from Figure 3.1, it is also not clear what it means to be the leader of a company. The dotted arrow indicates that the leader is the name of a person, but it is not clear whether that person is the CEO of the company, the owner, both, or neither.

The target instance that is presented in Figure 3.4(b) is an instance that has been generated under the assumption that we are interested in populating the target schema only with projects that have some government funding, and also, assuming that the leader of a company is its owner. Such an instance can be generated by the transformation query presented in Figure 3.5. The figure indicates the size and the complexity of the translation. Even for a simple transformation like the one in the example, the translation may be quite complicated.

3.2 The Nested Relational Data Model

In the example described in the previous section, both schemas were XML Schemas. Due to the heterogeneity that appears frequently between data sources, the data models in which the two schemas are expressed may be different. For example, one source may have relational data while another may have XML data. For such cases, we need to have a common platform on which to apply our theory. We choose to use a nested-relational data model as a common model to which we translate schemas from their native data source models. This model is the well-studied relational model [Cod90] extended to support nested sets in order to capture the nested nature of XML. It also supports nested referential integrity constraints like those present in XML Schemas [W3C01]. Adding order is one of the extensions of the model that we have not considered in this work, but is one of the future steps in order to capture order in XML data.

```
LET $doc0 := document("input XML file goes here")
RETURN
<T>{distinct-values (
   FOR $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/private, $x2 IN $doc0/S/project,
       $x3 IN $doc0/S/contact
   WHERE
      $x2/name/text() = $x0/project/text() AND
      $x1/text() = $x3/cid/text()
   RETURN
      <govProject>
         <code>  {  $x0/project/text() }  </code>
         <year>  {  "Sk295(", $x0/project/text(), ")" }  </year>
         {distinct-values (
         FOR $x0L1 IN $doc0/S/grant, $x1L1 IN $x0L1/sponsor/private, $x2L1 IN $doc0/S/project,
          $x3L1 IN $doc0/S/contact
         WHERE
            $x2L1/name/text() = $x0L1/project/text() AND $x1L1/text() = $x3L1/cid/text() AND
            $x0/project/text() = $x0L1/project/text()
         RETURN
            <funding>
               <fid>  {  $x0L1/gid/text() }  </fid>
               <pi>  {  $x0L1/pi/text() }  </pi>
               <finId>  {"Sk289(",$x3L1/phone/text(),",",$x0L1/budget/text(), ", ",
                     $x0L1/pi/text(), ", ", $x0L1/gid/text(), ", ",
                     $x0L1/project/text(), ")" }  </finId>
            </funding> )
         }
      </govProject> ) }
   {distinct-values (
   FOR $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/private, $x2 IN $doc0/S/project,
       $x3 IN $doc0/S/contact
   WHERE $x2/name/text() = $x0/project/text() AND $x1/text() = $x3/cid/text()
   RETURN
      <finance>
         <finId>  {  "Sk289(", $x3/phone/text(), ", ", $x0/budget/text(), ", ",
                  $x0/pi/text(), ", ", $x0/gid/text(), ", ", $x0/project/text(), ")" }  </finId>
         <amount>  {  $x0/budget/text() }  </amount>
         <sponsorPhone>  {  $x3/phone/text() }  </sponsorPhone>
         <company>  {  "Sk285(", $x3/phone/text(), ", ", $x0/budget/text(), ", ",
                  $x0/pi/text(), ", ", $x0/gid/text(), ", ", $x0/project/text(), ")" }  </company>
      </finance> ) }
   {distinct-values (
   FOR  $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/private,
        $x2 IN $doc0/S/project, $x3 IN $doc0/S/contact
   WHERE $x2/name/text() = $x0/project/text() AND  $x1/text() = $x3/cid/text()
   RETURN
      <company>
         <cname>  {  "Sk285(", $x3/phone/text(), ", ", $x0/budget/text(), ", ",
                    $x0/pi/text(), ", ", $x0/gid/text(), ", ", $x0/project/text(), ")" }  </cname>
         <leader>  {  "Sk288(", $x3/phone/text(), ", ", $x0/budget/text(), ", ",
                    $x0/pi/text(), ", ", $x0/gid/text(), ", ", $x0/project/text(), ")" }  </leader>
      </company> ) }
   {distinct-values (
   FOR  $x0 IN $doc0/S/company, $x1 IN $doc0/S/person, $x2 IN $doc0/S/person
   WHERE  $x1/SSN/text() = $x0/CEO/text() AND $x0/owner/text() = $x2/SSN/text()
   RETURN
      <company>
         <cname>  {  $x0/cname/text() }  </cname>
         <leader>  {  $x1/name/text() }  </leader>
      </company> ) }
</T>
```

Figure 3.5: A data translation query expressed in XQuery.

3.2.1 Types and Values, Schemas and Instances

Each value in our model has a type. A *type* τ is defined by the grammar:

$$\tau ::= \mathsf{Atomic} \mid \mathsf{Set\ of}\ \tau \mid \mathsf{Rcd}[l_1{:}\tau_1, \ldots, l_n{:}\tau_n] \mid \mathsf{Choice}[l_1{:}\tau_1, \ldots, l_n{:}\tau_n].$$

We assume the existence of an infinite set E of names. Let Atomic represent a basic set of atomic data types such as String, Integer, Date, etc., and let $dom(\mathsf{Atomic})$ be the union of the (possibly infinite) pairwise disjoint domains of values for every type in Atomic. Each basic atomic type τ_b is extended by adding a set O_b of *object identifiers*. Thus, in our model, the domain of each atomic data type $\tau \in \mathsf{Atomic}$ will be $dom(\tau) \cup O_b$. The object identifiers will be used in Chapter 4 in order to generate values for the parts of the target schema for which no value can be determined from the source schema.

Non-atomic types are *set* and *complex* types. A type Set of τ where τ is a complex type, is a collection type. A value of type Set of τ is actually a set object identifier and not a set in a traditional sense. Each type Set of τ has an infinite domain $O_{\mathsf{Set\ of}}$ of such set identifiers. Then, each set identifier o in $O_{\mathsf{Set\ of}}$ is associated with an unordered set $\{v_1, v_2, \ldots, v_n\}$ of values, each being of type τ. The association between each such value v_i and the set identifier o will be denoted by $v_i \in o$ and the association of the set identifier with all the values by $o(v_1, v_2, \ldots, v_n)$. This representation of sets (using set oids) is used to faithfully capture the nested nature of XML Schema instances. In the nested constraints presented in a following section, whenever we state the equality of two values that are of set type, we will mean equality of the two set oids (and not of their sets of children).[1]

Complex types are the *records* (Rcd) and *union* (or *choice*) (Choice) types. A value of type $\mathsf{Rcd}[l_1{:}\tau_1, \ldots, l_n{:}\tau_n]$ is an unordered tuple of pairs $[l_1{:}v_1, \ldots, l_n{:}v_n]$ where v_i is a value of type τ_i with $1 \leq i \leq n$. A value of type $\mathsf{Choice}[l_1{:}\tau_1, \ldots, l_n{:}\tau_n]$ on the other hand, is a pair $l_i{:}v_i$ where v_i is a value of type τ_i with $1 \leq i \leq n$. The symbols $l_1 \ldots l_n$ are referred

[1] The reason will become clear when we talk about how data generated by different mappings is merged to form one single instance of the target schema.

to as *labels* or *attributes*, and are from the set E of names.

The types τ_i and τ in the above specification of a complex and a set type respectively are said to be *directly used* in the complex or the set type. Similarly, the values v_i and v in the complex and set values above are said to be directly used by them, respectively.

A *schema* is a set of labels (called roots), each with an associated type. For example, the symbols `S` and `T` in Figure 3.1 are such roots for the source and target schema respectively. (A schema cannot contain two schema roots with the same label.) Any label-type pair that appears as a root or within the type of a root element is referred to as a *schema element* or simply *element*. Types within set types, e.g., the type Rcd[...] in the type Set of Rcd[...] are assumed to have the implicit and usually omitted label "ϵ". Due to the way types are specified, no cycles are allowed and a schema can be represented as a set of trees where each node represents a schema element. In the tree representation of a schema, the tree roots represent the schema roots. Hence, a schema can be modeled as a set of nodes and their parent-child relationships. In particular,

Definition 3.1 (Schema) *A* schema $\mathcal{S}$ *is a pair* $<\mathcal{E}, f^p>$ *where* $\mathcal{E}$ *is a multi-set of elements (i.e., label-type pairs, referred to as* schema elements*, and* f^p *is a total function* $f^p{:}\mathcal{E}\rightarrow\mathcal{E}\cup\{null\}$*, such that* $\forall el\in\mathcal{E}\ f^p(el)=el'$ *if element* el *is directly used in the type of* el'*, or* $f^p(el)=null$ *in which case element* el *is a* root element*.*

Example 3.2 *Figure 3.6 indicates the graphical representation of the source schema of Figure 3.1.* ∎

An instance is a set of label-value pairs. Due to the way values have been defined, an instance can be modeled as a set of trees.

Definition 3.3 (Instance) *An* instance $\mathcal{I}$ *is a pair* $<\mathcal{V}, f_v^p>$ *where* $\mathcal{V}$ *is a multi-set of label-value pairs, and* f_v^p *is total function* $f_v^p{:}\mathcal{V}\rightarrow\mathcal{V}\cup\{null\}$*, such that* $\forall v\in\mathcal{V}\ f_v^p(v)=v'$ *if label-value pair* v *is directly used in the value of the label-value pair* v'*, or* $f^p(el)=null$ *in which case the label-value pair* v *is called a* root*.*

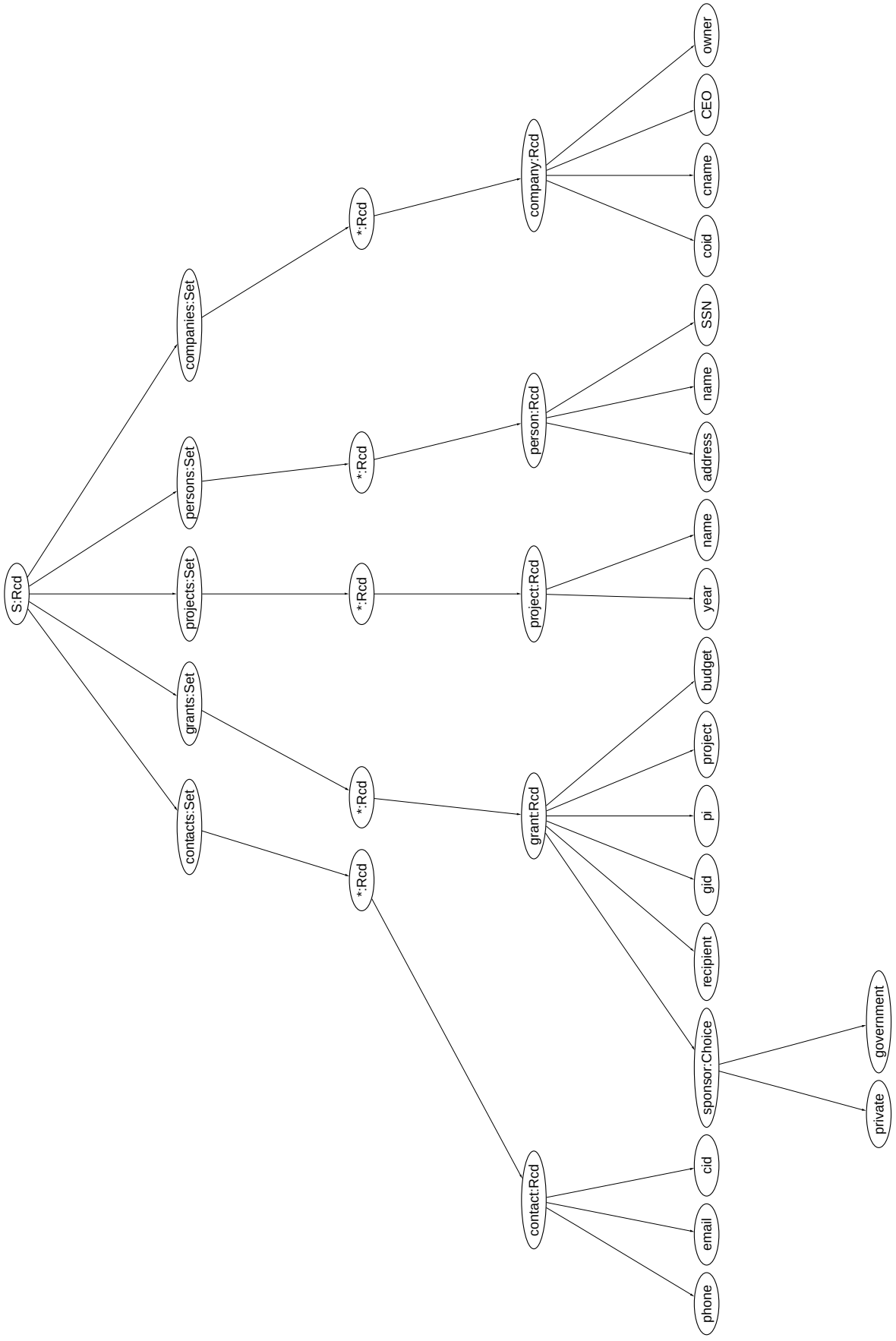


Figure 3.6: Graph representation of the source schema of Figure 3.1

Definition 3.4 (Conforms) *An instance $\mathcal{I}$* conforms *to schema $\mathcal{S}$ (in which case the instance is called a* valid instance *of schema $\mathcal{S}$), if there is a total injective function f_c that maps every label-value pair $l{:}v$ of the instance to a schema element $l{:}\tau$ of $\mathcal{S}$ such that value v is of type τ, and for each schema root $l_r{:}\tau_r$, there is at most one label-value pair $l_r{:}v_r$ for which $f_c(l_r{:}v_r) = l_r{:}\tau_r$. Given a valid instance of a schema, the* interpretation *of a schema element $l{:}\tau$, noted as $[[l{:}\tau]]$, is the set of pairs $l{:}v$ in the instance for which $f_c(l : v) = l : \tau$.*

We will usually refer to a label-value pair in an instance as a *value* if there is no risk of confusion.

Each data source has an instance and a schema to which the instance conforms. In order to distinguish between the different sources (or *databases*), each one is assigned a unique *database name*. When there is no risk of confusion, we may refer to a schema element $l{:}\tau$ by using only the name l, as we already do when we refer to schema elements of the schemas in Figure 3.1.

An alternative useful representation of a nested instance is through a labeled directed graph [PVM+02a], and in particular, a forest containing one tree for every value of the schema root that appears in the instance. The graph contains exactly one node for every value that appears in the instance. The relationship between a complex type value and the values of its attributes is denoted by an edge from the first to the second, labeled with the name of the attribute. The relationship between a set type value and the associated values in the set is also denoted by an edge from the first to the second that has the "*" label, referred to as an ϵ-edge.

Example 3.5 *The top of Figure 3.7 indicates a portion of the XML document of Figure 3.3 in the nested-relational model. The root element is the element* `S`*, which is of type record. Each attribute of that record is a set (represented by the set identifiers $O_1, \ldots, O_5$), that corresponds to the five top level set-type elements the source schema contains. Each*

row in the figure declares a complex value (record or union) and assigns it to one of these five sets. The lower part of Figure 3.7 depicts the graph representation of a part of the nested instance in the top of the figure. ∎

3.2.2 Queries and Schema Elements

We use queries to describe mappings, constraints, and represent schema elements. For queries, we adopt a *select-from-where* syntax. These three clauses use variables and expressions.

Definition 3.6 (Expression) *An* **expression** *e is defined by the grammar*

$$e \ ::= \ V \mid x \mid e.l \mid e{\rightarrow}l \mid f$$

where x is a variable, V is a schema variable, l is a record (or choice) label, e.l is a record projection, e→l is a union type choice, and f is a function call.

Functions return a single value or a set of values, and may accept as arguments one or more values (described as expressions).

Definition 3.7 (Expression Validity and Type) *Each expression has a type. Expression exp is valid and is of type τ if:*

- *it is a record projection e.l, and expression e is valid and of type* $\mathsf{Rcd}[\ldots, l{:}\tau, \ldots]$*, i.e., expression e is a record type that has an attribute with label l, or*
- *it is a union type choice e→l, and expression e is valid and of type* $\mathsf{Choice}[\ldots, l{:}\tau, \ldots]$*, i.e., expression e is a union type that has an attribute with label l, or*
- *it is a schema variable V and V:τ is a schema root, or*

S[projects:O_1, grants:O_2, contacts:O_3, companies:O_4, persons:O_5]

[project:[name:"DB2", year:"1996"]] $\in O_1$

[project:[name:"SQL Server", year:"1998"]] $\in O_1$

[project:[name:"Oracle", year:"1989"]] $\in O_1$

[grant:[gid:"g1", pi:"Bernstein", project:"SQL Server", budget:"10K",
recipient:"Microsoft", sponsor:[government:"NSF]]] $\in O_2$

[grant:[gid:"g2", pi:"Pirahesh", project:"DB2", budget:"8K",
recipient:"IBM", sponsor:[government:"NRC]] $\in O_2$

.....

[contact:[cid:"NSF", email:"info@nsf.gov", phone:"703-292-5111"]] $\in O_3$

[contact:[cid:"NRC", email:"info@nrc-cnrc.gc.ca", phone:"613-993-9101"]] $\in O_3$

.....

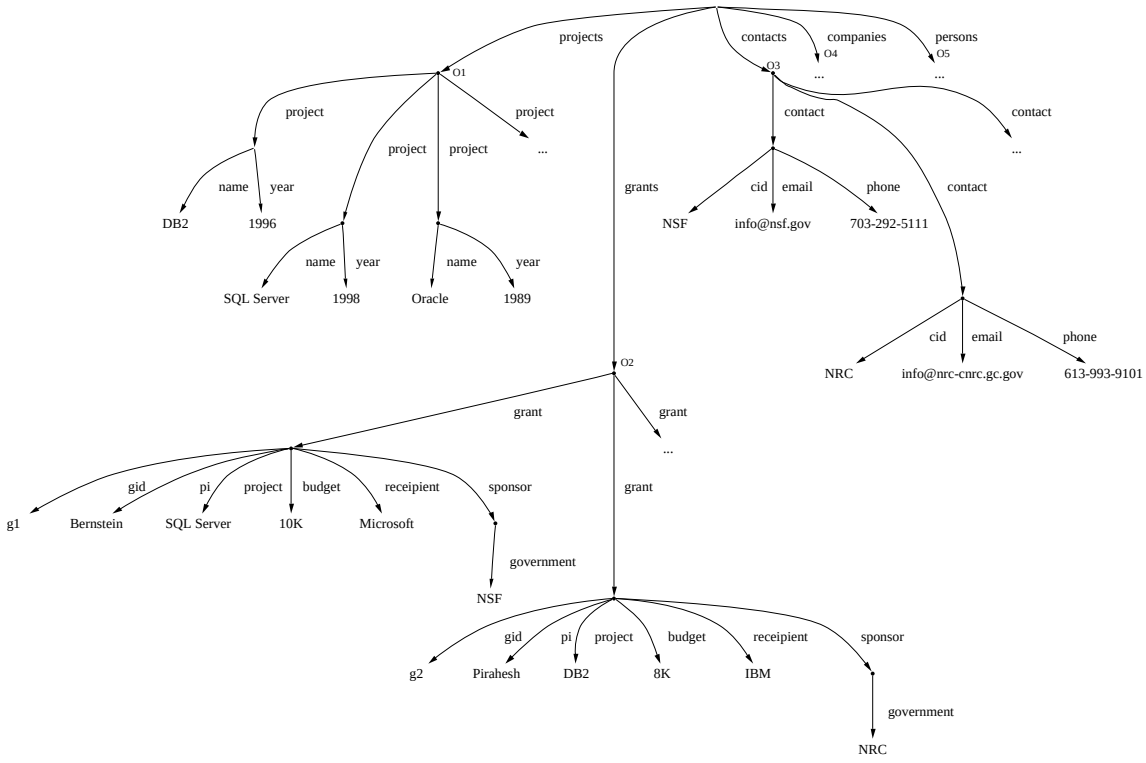


Figure 3.7: A portion of a nested relational instance and its graph representation.

- *it is a variable bound*[2] *to a valid expression of type* Set *of* τ*, or*
- *it is a variable bound to a valid expression of type* τ*, and* τ *is not a* Set *type, or*
- *it is a function call and the returned value is of type* τ*.*

An invalid *expression is an expression that is not valid. The type is not defined for invalid expressions.*

Each valid expression *refers* to a specific schema element. A schema variable expression V refers to the schema root with the label V. A valid expression $e.l$ refers to the element $l{:}\tau$, where the type of e is Rcd$[\ldots, l{:}\tau, \ldots]$. A valid expression $e{\rightarrow}l$, refers to the element $l : \tau$ where the type of e is Choice$[\ldots, l{:}\tau, \ldots]$. The schema element that a variable refers to depends on the expression to which the variable is bound. When a variable x is bound to a valid expression P, and P is of type Set of τ, then variable x iterates over the elements of type τ of the set, hence, it refers to element $*{:}\tau$. On the other hand, if variable x is bound to a valid expression P, and P is not of Set type, then variable x refers to the schema element to which the expression P refers to.

Definition 3.8 (Query) *A* **query** *is of the form:*

$$\begin{array}{l}\underline{\text{select}}\ e_0,\ e_1,\ \ldots,\ e_m\\ \underline{\text{from}}\ \ P_0\ x_0,\ P_1\ x_1,\ \ldots\ P_n\ x_n\\ \underline{\text{where}}\ c_0{=}c'_0\ \underline{\text{and}}\ c_1{=}c'_1\ \underline{\text{and}}\ \ldots\ \underline{\text{and}}\ c_k{=}c'_k\end{array}$$

where $x_0, \ldots, x_n$ *are variables and* $e_0, \ldots, e_m, P_0, \ldots, P_n, c_0, \ldots, c_k, c'_0, \ldots, c'_k$ *are valid expressions. A query is* **well-formed** *if, for every variable* y *used in an expression* e_i*,* c_i *or* c'_i*, there is a binding* P_k y *in the* from *clause, and for every variable* x_j *used in expression* P_i*,* $j < i$*. In addition, if variable* x_l *is used in expression* c_l*, and variable* $x_{l'}$ *in* c'_l*, then* $l{\geq}l'$*.*

[2]Variables are used in queries, hence, the variable binding will be defined when queries are introduced.

A *valuation* of an expression used in a query is the data value it describes when the variables of the query are instantiated to some values of an instance $\mathcal{I}$.

In this current work, we consider set semantics for our queries and data model. The study of multi-set semantics is one of our future research plans. We will also restrict the syntax of the queries we consider to those that bind variables only to set type expressions and union type choices. We will call queries that satisfy this condition *canonical* queries.

Definition 3.9 (Canonical Query) *A* **canonical** *query is a well-formed query for which every expression P_i in the* from *clause is either a set type expression or an expression of the form $e \rightarrow l$. In addition, expression P_i does not contain a sub-expression that is of set type or of the form $e \rightarrow l$. Furthermore, every expression in the* select *or the* where *clause is of atomic type and is of the form described by the grammar $e ::= S|x|e.l|f$.*

Note that working with canonical queries does not restrict the expressiveness of the queries, since every non-canonical query can be rewritten into an equivalent canonical query by introducing new variables or removing others.

Example 3.10 *For the nested source schema of Figure 3.1, the query*

```
select  x→private
from    S.grants.grant.sponsor x
```

is not a canonical query since neither the sub-expression `S.grants`*, which is of set type, nor the choice $x \rightarrow$* `private` *have a variable bound to them. However, the query can be reformulated by introducing two new variable bindings* `S.grants` *g and g.*`grant.sponsor`*→*`private` *r in the* from *clause, removing the x binding, and finally, using variable r in the* select *clause instead of the expression $x \rightarrow$* `private`*. In short, the equivalent canonical query will be:*

```
select  r
from    S.grants g, g.grant.sponsor→private r
```

■

For the rest of this document, every reference to a query will mean a canonical query.

For a given binding $P\ x$, if expression P is of set type, then variable x will bind to the individual values of type τ in the set, and will be said to refer to element $*{:}\tau$. If expression P is a union type choice $e{\rightarrow}l$, then variable x will bind to the values of type τ under the choice l of e. Sometimes, the "*" symbol may be used in the select clause to denote all possible valid atomic type expressions.

The form of queries we consider is generic and can represent a significant subset of SQL and XQuery. Note that, although our queries have only one from and where clause, they can represent nested queries. This is done by "flattening". In particular, the from clause of a nested query can be appended to the from clause of the parent query and its where clause will be merged with the where clause of the parent query. Furthermore, even if the queries are restricted to return sets of tuples, the schemas may be nested schemas and may contain complex or abstract types. A complex (nested) output like those required by XML query languages [CCDF+01] may be achieved by combining more than one query as will be shown in the next chapter.

To understand and reason about queries (and later mappings), we need to be able to understand and represent relationships between queries. We use *renamings* (one-to-one functions) to express a form of query subsumption.

Definition 3.11 (Query Domination) *A query q_1 is* **dominated** *by a query q_2 (noted as $q_1 \dot{\preceq} q_2$) if there is a renaming (one-to-one function) h from the variables of q_1 to the variables of q_2 such that the* select *and* from *clauses of $h(q_1)$ are subsets, respectively, of the* select *and* from *clauses of q_2, and for every equality eq that appears in the* where *clause of q_1, $h(eq)$ either appears in the* where *clause of q_2 or is logically implied by it.*

Nesting and grouping are supported by using P_i expressions that include previously bound variables. Thus, nested conjunctive queries [PVM+02b], SQL, and FLWR expressions with no wildcards can be expressed. Note that the functionality of wildcards is not a real requirement in our queries, since, having the schema available, we can always replace them with explicit parts of the schema. The query language covers the core queries that have been considered in the literature for data exchange and integration applications [TH04, Len02, PVM+02b, FKMP03b], and can also be extended to include aggregation functions, negation and order.

Queries can be used to represent elements within schemas. A schema element can be identified by the query used to retrieve all the instance values in the interpretation of the element.

Definition 3.12 (Schema Element Query) *A* **schema element query** *is a canonical query:*

$$\underline{\text{select}}\ e_{n+1}\ \underline{\text{from}}\ P_0\ x_0,\ P_1\ x_1\ ,\ \ldots\ P_n\ x_n$$

where each P_k with $k \geq 1$ uses variable x_{k-1}, P_0 uses a schema root and expression e_{n+1} uses variable x_n. If the from *clause is empty, e_{n+1} starts at a schema root. When the details of the* from *clause are unimportant, the schema element can be noted as* select e from P. *A schema element may also be expressed relative to another schema element* select e' from P'. *In that case, P_0 starts at expression e' instead of the schema root.*

Example 3.13 *For the source schema in Figure 3.1, the schema elements* `amount` *and* `private` *under* `grant` *can be described by the following two queries, respectively:*

a_1: select g.`grant.amount`
from `S.grants` g
a_2: select r
from `S.grants` g, g.`grant.sponsor`→`private` r

Notice that, since expression S.grants *is of type* Set of Rcd[*grant* : Rcd[...]], *variable g in query* a_1 *is bound to the record type values of the set (i.e.,* Rcd[*grant* : Rcd[...]]*). Hence, in order to get element* amount*, variable g has to be first record-projected on* grant *and then on* amount*. On the other hand, since expression g*.grant.sponsor→private *is a choice selection, variable s in query* a_2 *is bound to the atomic type values described by the expression g*.grant.sponsor→private*, hence, variable s is used in the* select *clause as is.* ■

For the rest of the document, the terms "schema element" and "schema element query" will be considered equivalent and will be used interchangeably. However, if in some case the distinction needs to be made, we will state this explicitly.

3.2.3 Schema Constraints and Mappings

For *schema constraints*, we consider a very general form of referential constraints called *nested referential integrity constraints (NRIs)* [PVM+02b] extended to support choice types. NRIs capture naturally relational foreign key constraints as well as the more general XML Schema *keyref* constraints. The simplest form of NRI relates two schema elements and represents an inclusion constraint between them.

In its simplest form an NRI consists of three parts. The first one is characterized by the keyword **foreach**, the second by the keyword **exists** and the third by the keyword **with**. Given the two schema elements that the NRI needs to associate, the three parts of the NRI are constructed as follows. The **foreach** part is the from clause of the first element, the **exists** part is the from clause of the second, and the **with** part is an equality operator between the select clauses of the two elements. In the general case of an NRI:

Definition 3.14 (Unary NRI Constraint) *Given two schema elements with element queries* select e_1 from P_1 *and* select e_2 from P_2*, respectively, both defined relative to a*

schema element with element query select e_0 from P_0, *an* **NRI constraint** *is a statement of the form:*

foreach P_0 [**foreach** P_1 **exists** P_2 **with** C]

where C *is the equality* e_2=e_1. *The element* select e_0 from P_0 *is referred to as the* **context element** *of the constraint. If the context element is a schema root, then the* **foreach** P_0 *part can be omitted.*[3] *part from.*

The above definition of a constraint can be naturally extended to associate multiple elements (n-ary NRI). An NRI **foreach** X **exists** Y **with** C can associate a list X of n atomic elements with a list Y of n atomic elements. The **with** clause will be a conjunction of n equalities. Note that the elements of Y may be denoted relative to some variable of X, or to a schema root.

In the rest of the document, we will consider NRIs that associate only two elements, but the algorithms that will be presented apply equally well to the case where we have NRIs associating more than two elements. The implementation that we have allows for n-ary NRIs.

Example 3.15 *The foreign key* f_2 *on the source schema of Figure 3.1 is expressed as follows:*

f_2: **foreach** S.grants g, g.grant.sponsor→private r
exists S.contacts c
with c.contact.cid=r

Notice that NRIs can be expressed using other kinds of representation that are common in the literature, for example, 1st order logic [AHV95]. Our internal implementation representation, for example, follows the representation shown below:

[3]The symbol ']' in definition 3.14 is used only to indicate the part of the NRI that does not determine the context element.

$$f_2 : \forall\ g \in \texttt{S.grants},\ r \in g.\texttt{grant.sponsor} \rightarrow \texttt{private} \Rightarrow \exists\ c \in \texttt{S.contacts},$$

$$c.\texttt{contact.cid} = r$$

■

Figure 3.8 provides a graphical presentation of the dependencies that have been considered in the literature. NRIs are in the volume on the right of the border between tgds and edgs.

A mapping is an expression that specifies how the data in one schema is related to the data of a second. We consider a very general form of mapping that subsumes a large class of mappings used in a variety of applications. As described in Section 2.1, a *mapping* m from a Schema $\mathcal{S}$ (called the *source* schema) to schema $\mathcal{T}$ (called the *target* schema) is an assertion of the form: $Q^{\mathcal{S}} \rightsquigarrow Q^{\mathcal{T}}$, where $Q^{\mathcal{S}}$ is a query over $\mathcal{S}$ and $Q^{\mathcal{T}}$ is a query over $\mathcal{T}$ [Len02]. In particular, the meaning of such a mapping is that, for all such instances, for each tuple generated by executing query $Q^{\mathcal{S}}$ over the source schema, the same tuple must appear in the answer set of the query $Q^{\mathcal{T}}$ if it gets executed over the target schema. Such mappings can be seen as tuple-generating dependencies from schema $\mathcal{S}$ to schema $\mathcal{T}$ [FKMP03b], and can also be expressed as NRI constraints. Since both sides can be queries (and actually nested conjunctive queries), they are expressive enough to naturally include a large number of mappings that are met in practice, and the mappings that have most commonly been considered in the integration research literature.

If the queries $Q^{\mathcal{S}}$ and $Q^{\mathcal{T}}$ are restricted to (type compatible) queries that return sets of tuples and the relation $\rightsquigarrow$ is the subset-or-equals relation $\subseteq$, then such mappings are called *sound* mappings [Len02]. Potential type incompatibilities can be resolved through type transformation functions. The form of such mappings is very general and includes as special cases the GAV (global-as-view) [BLN86] and LAV (local-as-view) [LRO96] views used in data integration systems or the GLAV (global-and-local-as-view) mappings [FLM99] used in transforming data between independent schemas [PVM+02b], in peer-to-peer query answering [MH03] and in data exchange [FKMP03b].

Definition 3.16 (Mapping) *A mapping is an expression of the form*

$$\textbf{foreach}\ Q^{\mathcal{S}}\ \textbf{exists}\ Q^{\mathcal{T}}$$

where $Q^{\mathcal{S}}$ and $Q^{\mathcal{T}}$ are queries on the source and the target schemas, respectively, with the same number of type compatible expressions in their select *clauses.*

An alternative representation of a mapping that emphasizes its nature as an inter-schema constraint is one in which the pairs of expressions in the select clauses are explicitly mentioned in a **with** clause [YP04]. In particular, a mapping "**foreach** select X_s from X_{fw} **exists** select Y_s from Y_{fw}" can be expressed in the form "**foreach** select $*$ from X_{fw} **exists** select $*$ from Y_{fw} **with** C" where C is a conjunction of equalities. Each equality is between an expression in X_s and the Y_s expression in the corresponding position. For consistency with the NRI constraint representation, the "select *" part will usually be dropped.

Example 3.17 *Consider a mapping that retrieves from the source schema of Figure 3.1 the names of companies along with the name of their owner, and generates catalog entries in the target schema with the name of the company and the name of the owner as a leader.*

```
foreach
 select  w.company.cname, n.person.name
 from    S.companies w, S.persons n
 where   n.person.SSN=w.company.owner
exists
 select  o.company.cname, o.company.leader
 from    T.companies o
```

Using a **with** *clause, the above mapping becomes:*

```
foreach  S.companies w, S.persons n
 where   n.person.SSN=w.company.owner
exists   T.companies o
with     o.company.cname=w.company.cname and
         o.company.leader=n.person.name
```

■

We will interpet mappings as inclusion dependencies between a source and a target schema instance. The semantics of a mapping are the semantics of an inclusion dependency.

Definition 3.18 (Mapping Satisfaction) *Given a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, the mapping m :* **foreach** $Q^{\mathcal{S}}$ **exists** $Q^{\mathcal{T}}$ *is said to be satisfied by the instances $\mathcal{I}^{\mathcal{S}}$ and $\mathcal{I}^{\mathcal{T}}$ of schemas $\mathcal{S}$ and $\mathcal{T}$, respectively, if the results of applying query $Q^{\mathcal{S}}$ on instance $\mathcal{I}^{\mathcal{S}}$ are included in the results of applying query $Q^{\mathcal{T}}$ on instance $\mathcal{I}^{\mathcal{T}}$.*

Note that although the "units" whose exchange is described by a mapping are tuples of atomic values, this does not mean that complex type values, e.g., a complex XML element, cannot be exchanged. Complex values can be exchanged, but their structure must be specified explicitly by the queries that form the mapping. We describe this specification in Section 4.4.

A mapping from a schema $<\mathcal{E}_1,f_1^p>$ to schema $<\mathcal{E}_2,f_2^p>$ can be modeled as a triple $<\mathcal{E}_s,\mathcal{E}_t,W_c>$ where the sets $\mathcal{E}_s$ and $\mathcal{E}_t$ consist of all the schema elements that are referred to by the expressions in the **foreach** and **exists** clauses of the mapping, respectively, and $\mathcal{E}_s \subseteq \mathcal{E}_1$, $\mathcal{E}_t \subseteq \mathcal{E}_2$, and W_c is a set of atomic type element pairs from $(\mathcal{E}_s \cup \mathcal{E}_t) \times (\mathcal{E}_s \cup \mathcal{E}_t)$. The set W_c consists of pairs of elements that are referred to either by two expressions in a binary predicate in a where clause (in which case both elements are from the same schema), or by two expressions in a binary predicate of the **with** clause.

Since mappings are based on queries, the notion of dominance can be naturally extended to mappings.

Definition 3.19 (Dominance of Mapping) *Mapping m_1: "***foreach** A_1 **exists** A_1' **with** W_1*" is* dominated *by mapping m_2: "***foreach** A_2 **exists** A_2' **with** W_2*" (denoted*

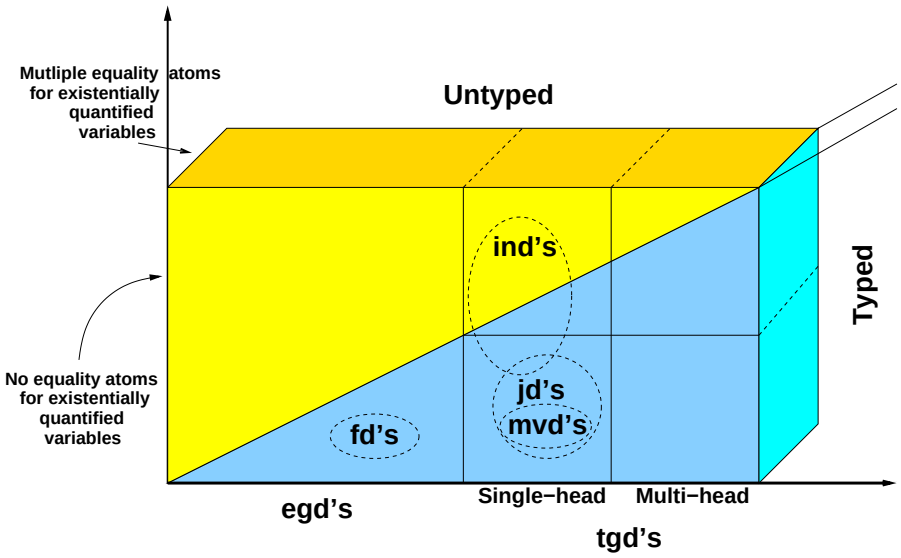


Figure 3.8: A classification of the various types of dependencies. NRIs are tgds.

as $m_1 \dot{\preceq} m_2$*) if* $A_1 \dot{\preceq} A'_1$*,* $A_2 \dot{\preceq} A'_2$*, and for every condition* $e\theta e'$ *in* W_1*,* $h_1(e)\theta h_2(e')$ *is in* W_2 *(or implied by* W_2*), where* h_1 *and* h_2 *are the renaming functions from* A_1 *and* A'_1 *to* A_2 *and* A'_2 *respectively.*

Finally, XML Schema ID and IDREF can be modeled as atomic type elements with a set of XML *key* and *keyref* constraints.

3.3 Translating Models to the Nested Relational Model

Before being able to apply our theory, we need to translate the schemas into a nested relational representation. This means translating the structure of the schema into a set of nested types and the schema constraints into a set of NRIs.

We consider relational schemas consisting of a set of tables and a set of foreign key constraints. A table contains a set of attributes. To translate a relational schema into its nested relational representation, a schema root is created in order to represent the database schema as a whole. Although our algorithms will work even if we do not do

this, we use this approach for two reasons. One is to be compatible with the XML representation that requires XML documents to have a single root. The second and most important reason is that it allows us to refer to various tables of different data sources, even if they have the same name. For instance, the table R of the database DB_1 and the table R of DB_2 will have no conflict since they will appear under the schema roots that correspond to the database DB_1 and DB_2, respectively. Each schema root will be of Rcd type having one attribute for each relational table in the relational schema. Each such attribute will be of type Set of Rcd$[TNAME$: Rcd$[...]]$ where $TNAME$ is a name that is created to represent a tuple of the table. As shown, the $TNAME$ attribute will be of Rcd type and will have as many attributes as the number of columns in the relational table it represents. Each such attribute will have the column name of the table as a label and will be of the same type as the type of the column.

To emphasize the relationship between the relational model and its nested relational representation, every schema root of type Set of Rcd[...] with atomic type attributes in the record will be referred to as a *relation* and every record value in such a set will be referred to as *tuple*.

In order to deal with the case where some attributes can be null (or optional in the case of XML Schemas that will be presented shortly), the attribute (or label) of a record type has been extended to carry this extra information. A nullable attribute will be represented by appending the "#" symbol at the end of its label name and an optional attribute by appending the "?" symbol.

Example 3.20 *Consider a relational schema of a database* DB *that is defined by the following DDL statements:*

```
CREATE TABLE Employees AS (
       name         varchar(20) not null,
       address      varchar(50),
       salary       int not null,
       department   varchar(20) not null,
       FOREIGN KEY  (department) REFERENCING Department
       )

CREATE TABLE Departments AS (
       id           varchar(20) not null,
       employees    int not null,
       name         varchar(40) not null,
       contact      varchar(35),
       PRIMARY KEY  (id)
)
```

These two tables can be represented as a nested relational schema as follows:

```
DB: Rcd [
   Employees: Set of Rcd [
      Employee: Rcd [
         name: String,
         address#: String,
         salary: Integer,
         department: String ] ]
   Departments: Set of Rcd [
      Department: Rcd [
         id: String,
         personel: Integer,
         name: String,
         contact#: String ] ]
```

Notice that the attributes `address` *and* `contact` *end with the # symbol to indicate that they can be null. Another important observation is that the terms* `Employee` *and* `Department` *do not appear in the relational DDL statements. They form the name that is given to the tuples of the relations* `Employees` *and* `Departments`*, respectively. In a similar way, the element* `sponsor` *of the source schema of Figure 3.1 can be seen as the nested relational representation of a relational database that consists of the following relations:*

```
Projects(name, year),
Grants(gid, pi, project, budget, recipient)
Contacts(cid, name, phone)
Companies(coid, cname, CEO, owner)
Persons(SSN, name, address)
```

■

A foreign key constraint is represented in the nested model as an integrity constraint where the **foreach** part specifies the attribute that is the foreign key, and the **exists** part specifies the key attribute that is referenced.

Example 3.21 *The foreign key constraint from the* `department` *attribute of* `Employees` *relation to the* `id` *attribute of relation* `Departments` *is represented as follows:*

foreach `DB.Employees` e
exists `DB.Departments` d
with e`.Employee.department=`d`.Department.id`

■

An XML Schema [W3C01] consists of a set of elements, attributes, and type definitions. An element (attribute) definition associates an element (attribute) name with a type. Types[4] can be *atomic*, *simple*, or *complex*. A *complex* type is a tree. Internal nodes of the tree are always *model group*[5] nodes. The leaves of the tree are nodes labeled with element names. Each such node has a quadruple (*maxOccurence*, *minOccurence*, *isNullable*, *isOptional*). The first two are non-negative integers that determine the cardinality of the element. The next two are Boolean and specify whether the element value can be null and whether it is mandatory.

With respect to XML Schema, Set types are used to model repeatable elements (or repeatable groups of elements). For a repeatable element `X` a new Set of Rcd[`X`:...] type

[4] Attributes cannot be of complex type.
[5] Model group nodes are the *all, sequence* and *choice*.

is generated where `X` is the name of the element. A notable inconsistency between the nested relational model and XML Schema is that, in an XML Schema, one may have sets that cannot be referenced. For example, in the XML Schema of Figure 3.2, one can notice the repeatable element `project`. This means that a new type Set of Rcd[`project`:...] will be generated to model it. This set type needs to be placed under the Rcd type element `S`. Due to the way a record type has been defined in the nested relational model, each of its attributes should have a label. Unfortunately, such a label is not specified by the XML schema. Thus, in order for the type Set of Rcd[`project`:...] to be placed under the record type of `S`, a new label name has to be invented. The `projects` element is the invented label for this particular example.

Example 3.22 *Looking at the XML Schema of Figure 3.2, note that there is no element* `projects`*. We will refer to such "invented" attribute labels as* **virtual** *elements. In the nested schemas of Figure 3.1, the* virtual *elements have been underlined.* ■

The Rcd and Choice are used to represent the "all" and "choice" *model groups* [W3C01] respectively. Order is not considered. A Set type represents an unordered set and an XML Schema "sequence" is modeled the same way "all" is modeled, i.e., through a Rcd type.

The syntax of an XML Schema can be formally described by the following grammar:

τ_{root}	::=	τ_c	*A root is a complex type*
τ_c	::=	all $p_1\ p_2\ \ldots p_n$	//<all> <$element_1$/> ... <$element_N$/> </all>
		| sequence $p_1\ p_2\ \ldots p_n$	//<sequence> <$element_1$/> ... <$element_N$/> </sequence>
		| choice $p_1\ p_2\ \ldots p_n$	//<choice> <$element_1$/> ... <$element_N$/> </choice>
		| $\tau_c{}^*$	//<element maxOccurs="unlimited"/>
ρ	::=	all $p_1\ p_2\ \ldots p_n$	//<all> <$element_1$/> ... <$element_N$/> </all>
		| sequence $p_1\ p_2\ \ldots p_n$	//<sequence> <$element_1$/> ... <$element_N$/> </sequence>
		| choice $p_1\ p_2\ \ldots p_n$	//<choice> <$element_1$/> ... <$element_N$/> </choice>
		| ρ^*	//<element maxOccurs="unlimited"/>
		| $A{:}\tau$	//<element name="A" type =" τ" >

Based on the above grammar, the translation of an XML Schema into a nested relational representation can be formally specified by the following rules:

$$\begin{aligned}
\mathcal{R}[\tau_c ::= \tau_c^*] &= \mathsf{Set\ of}\ \mathcal{R}[\tau_c] \\
\mathcal{R}[\rho ::= \rho^*] &= \mathsf{Set\ of}\ \mathcal{R}[\rho] \\
\mathcal{R}[\rho ::= A{:}\tau] &= A : \mathcal{R}[\tau] \\
\mathcal{R}[\tau_c/\rho ::= \mathsf{all/sequence}\ \rho_1\ \ldots\ \rho_n] &= \mathsf{Rcd}[\mathcal{F}[\rho_1],\ \ldots,\ \mathcal{F}[\rho_n]] \\
\mathcal{R}[\tau_c/\rho ::= \mathsf{choice}\ \rho_1\ \ldots\ \rho_n] &= \mathsf{Choice}[\mathcal{F}[\rho_1],\ \ldots,\ \mathcal{F}[\rho_n]] \\
\mathcal{F}[\rho^*] &= A_{virtual} : \mathcal{R}[\rho^*] \\
\mathcal{F}[\mathsf{all/sequence}\ \rho_1\ \ldots\ \rho_n] &= A_{virtual} : \mathcal{R}[\mathsf{all/sequence}\ \rho_1\ \ldots\ \rho_n] \\
\mathcal{F}[\mathsf{choice}\ \rho_1\ \ldots\ \rho_n] &= A_{virtual} : \mathcal{R}[\mathsf{choice}\ \rho_1\ \ldots\ \rho_n] \\
\mathcal{F}[A : \tau] &= \mathcal{R}[A : \tau]
\end{aligned}$$

For some specific XML Schema structures, the above rules may generate a number of nested relational schema elements during the translation to the nested relational model, without actually needing them. For such cases, an optimization step is required. Informally speaking, the optimization step tries to eliminate virtual attributes that were created as a result of a repeatable element, if they are not absolutely needed. The optimization step can be described through the following two rules:

$$\mathsf{Rcd}[A_{virtual} : \tau_c[\mathsf{all/sequence/choice}\ \rho_1\ \ldots\ \rho_n]] \rightarrow \tau_c[\mathsf{all/sequence/choice}\ \rho_1\ \ldots\ \rho_n]$$

and

$$\mathsf{Rcd}[A_{virtual} : \tau_c[\rho^*]] \rightarrow \tau_c[\rho^*]$$

Example 3.23 *As an example of such a case, consider the portion of an XML Schema that describes a team of authors:*

```
<element name=''authorteam''>
   <all>
      <element name=''author'' maxOccurs=''*'' type=''T''/>
   </all>
</element>
```

Following the translation rules described before, the resulting nested relational representation will be:

```
authorteam : Rcd [Avirtual:Set of Rcd [ author : T]]
```

but, by following the optimization rules, it will give

```
authorteam : Set of Rcd [ author : T]]
```

■

3.4 Type Checking

The data model we use is a strongly typed model. Every variable or expression used in constraints and mappings has its own type. The type assignment is done through a type-checking process. The type-checker runs every time a schema of a source is wrapped into its nested relational representation or modified. Through this process, every variable or expression is annotated with its type. When a variable is assigned a type, this information needs to be stored for subsequent reference. This is the role of the *typing context* [Pie02].

Definition 3.24 (Context) *A* **typing context** Γ *(or simply* **context***) is a list of variables and their types. The "comma" operator extends* Γ *by concatenating a new binding, the one on the right of the comma.*

To avoid confusion between a new binding and any binding that may appear in a context Γ, we require that the name x of each variable be distinct from the variables bound by the context Γ. Since variables may be renamed without affecting any results, this condition can always be satisfied. A context, can thus be thought of as a finite function from the variables to their types. A interesting implementation of a context (which is also followed in our implementation) is to use a DeBrujin or Global index [Pie02] in order to refer to a variable in the context. This means that each variable is uniquely identified by its position in the context, eliminating the need for having variable names,

1 $$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

2 $$\frac{\Gamma \vdash e : \text{Rcd}[l_1{:}T_1,...,l_n{:}T_n]}{\Gamma \vdash e.l_j : T_j}$$

3 $$\frac{\Gamma \vdash e : \text{Choice}[l_1{:}T_1,...,l_n{:}T_n]}{\Gamma \vdash e{\rightarrow}l_j : T_j}$$

4 $$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash bin(e_1,e_2) : bool}$$

5 $$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \text{ AND } e_2 : bool}$$

6 $$\frac{\Gamma \vdash C : bool \quad \Gamma \vdash W : bool}{\Gamma \vdash \text{construct } () \text{ where } C \text{ with } W : bool}$$

7 $$\frac{\Gamma \vdash e : \text{Set of } T \quad \Gamma,\ x : T \vdash \text{construct } G \text{ where } C \text{ with } W : bool}{\Gamma \vdash \text{construct } e\ x,\ G \text{ where } C \text{ with } W : bool}$$

8 $$\frac{\Gamma \vdash e : \text{Choice}[l_1{:}T_1,...,l_n{:}T_n] \quad \Gamma,\ x : T_k \vdash \text{construct } G \text{ where } C \text{ with } W : bool}{\Gamma \vdash \text{construct } e{\rightarrow}l_k\ x,\ G \text{ where } C \text{ with } W : bool}$$

9 $$\frac{\Gamma \vdash C_1 : bool \quad \Gamma \vdash \text{construct } H \text{ where } C_2 \text{ with } W : bool}{\Gamma \vdash \text{foreach } () \text{ where } C_1 \text{ construct } H \text{ where } C_2 \text{ with } W : bool}$$

10 $$\frac{\Gamma{\vdash}e{:}\text{Set of } T \quad \Gamma,x{:}T{\vdash}(\text{foreach } G \text{ where } C_1 \text{ construct } H \text{ where } C_2 \text{ with } W){:}bool}{\Gamma{\vdash}(\text{foreach } e\ x,\ G \text{ where } C_1 \text{ construct } H \text{ where } C_2 \text{ with } W){:}bool}$$

11 $$\frac{\Gamma{\vdash}e{:}\text{Choice}[...,l_k{:}T_k,...] \quad \Gamma,x{:}T_k{\vdash}(\text{foreach } G \text{ where } C_1 \text{ construct } H \text{ where } C_2 \text{ with } W){:}bool}{\Gamma \vdash (\text{foreach } e{\rightarrow}l_k\ x,\ G \text{ where } C_1 \text{ construct } H \text{ where } C_2 \text{ with } W){:}bool}$$

Table 3.1: Typing rules for queries, mappings and constraints

and allowing the context to be implemented as a simple list instead of an associative array.

The type checker works as follows. When a schema is turned into its nested relational representation, for every schema root, a variable binding is appended in the context for the schema root and its associated type. When a schema constraint is read or a new mapping is constructed, its NRI representation is formed. The type checker processes the NRI by processing first its **foreach** part, then its **exists**, and finally its **with** part. The variable bindings are processed in the order in which they appear. For every variable binding $e\ v$, of a variable v to expression e, the expression e is type-checked first and its type is assigned to the variable. A new binding is then created in the context Γ for variable v and a type τ, if expression e is of type Set of τ, or type τ' if expression e is a union type choice expression of type τ'. When an equality condition in a where or **with** clause is processed, its left and right expressions are type-checked first, and then a check is performed to ensure that the two types are the same.

In general, the type checking process can be described by the typing rules of Table 3.1, where Γ is a context, T is a type and x is a variable of the nested relational model. The notation in that figure is the standard notation used in describing typing mechanisms [Pie02]. Each rule consists of two parts. For the part below the line to be satisfied, the part that is above the line needs to be satisfied.

Once the source and the target schema have been translated to the nested relational model, the constraints represented as NRIs, and type-checked, the mapping generation process can begin.

Chapter 4

Mapping Generation

The previous chapter introduced the problem of managing mappings and demonstrated how complicated the mappings can become. This chapter introduces a novel framework for the generation of mappings between any combination of relational and XML Schemas, in which high-level, user-specified correspondences are translated into semantically meaningful queries that transform source data into a target representation. The proposed framework works in three main phases (summarized in Figure 4.1). In the first phase, the high-level correspondences, expressed as a set of element-to-element inter-schema relationships, are converted into a set of mappings that capture the design choices made in the source and target schemas (including their hierarchical organization and the grouping of elements into nested structures and sets, as well as their nested referential constraints). Section 4.1 formally introduces the correspondences and explains the reasons why correspondences were chosen as the high level specification of how the elements of the two schemas correspond. Section 4.1 shows how we can discover semantically related schema elements. Section 4.2 shows how the semantic relationships between schema elements are captured in the schema structure and constraints. Section 4.3 describes the mapping generation process of Figure 4.1 by showing how these semantic relationships can be converted to semantically complete logical mappings. The mapping generation

algorithm produces a set of mappings that are consistent with the schema constraints. For data exchange, a second phase is needed to translate these logical mappings into implementation-independent queries over the source schemas that are guaranteed to produce data satisfying the constraints and structure of the target schema, and preserve the semantic relationships of the source. Non-null target values may need to be created in this process to achieve this guarantee. This phase (box data translation in Figure 4.1) is presented in Section 4.4. The same section also describes the last phase of the process in which the generated queries are converted to the native query language of the data sources, i.e., SQL, XQuery, or XSLT. Section 4.5 studies the complexity and the properties of the mapping generation algorithm. Finally, Section 4.6 describes the implementation of the mapping generation algorithm in a tool, named Clio, and also describes our experience using Clio on several real schemas.

The work presented in this chapter has been published in VLDB [PVM+02b] and demonstrated in ICDE [PHV+02].

4.1 Correspondences

In order to shield users from the laborious task of manually generating mappings between two schemas, we must first understand how the elements of the two schemas correspond. We will use the output of a *schema matching* [RB01] process where the two schemas are compared and the output is a set of relationships between the elements of the two schemas. These "relationships" can be seen as high-level mapping specification between the schemas. There are numerous proposals for representing inter-schema relationships, including proposals for using general logic expressions to state how components of one schema correspond to components of the other [MZ98].

For this *high-level mapping specification*, we advocate the use of *element correspondences*, or *correspondences* for short, which map atomic elements of a source schema to

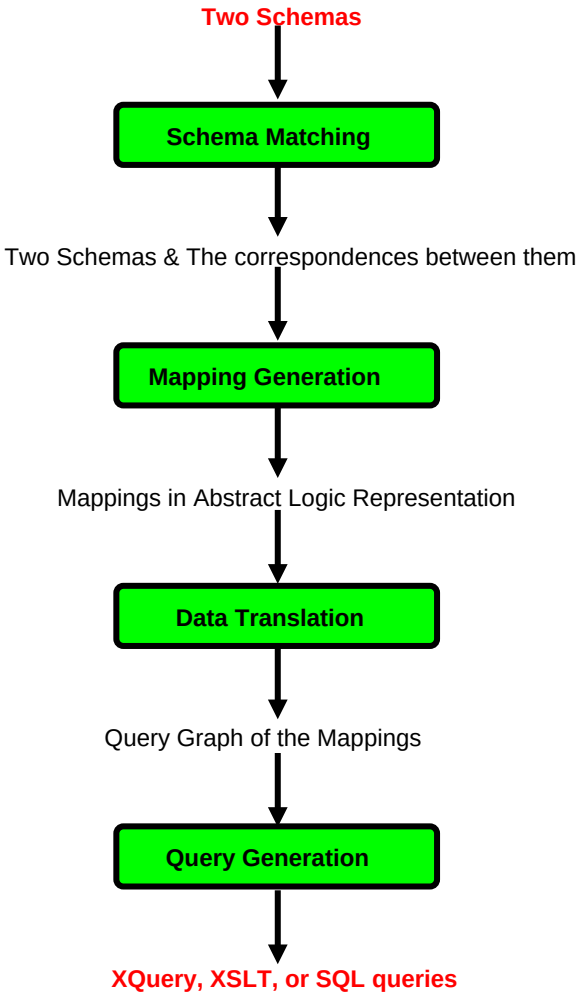


Figure 4.1: Overview of the mapping generation process.

atomic elements of a target. Intuitively, a correspondence is a pair of schema elements, one from the source and one from the target schema. This specification is independent of logical design choices such as the grouping of elements into tables (normalization choices), or the nesting of records or tables (for example, the hierarchical structure of an XML schema). In other words, one need not specify the logical access paths (join or navigation) that define the associations between the elements involved. Therefore, even users that are unfamiliar with the complex structure of the schema can easily specify them. Furthermore, no knowledge of any special language is needed for such specifications. Correspondences can be represented graphically through simple arrows between

the elements of the two schemas.

A correspondence can be formally defined as a special case of a mapping.

Definition 4.1 (Correspondence) *A* **correspondence** *from an element el_1 of a source schema to an element el_2 of a target, is a mapping:*

$$\begin{array}{ll} \textbf{foreach} & P_0\ x_0,\ P_1\ x_1\ ,\ \ldots\ P_n\ x_n \\ \textbf{exists} & P'_0\ x'_0,\ P'_1\ x'_1\ ,\ \ldots\ P'_n\ x'_n \\ \textbf{with} & e_2 = e_1 \end{array}$$

where the queries

$$\underline{\text{select}}\ e_1\ \underline{\text{from}}\ P_0\ x_0,\ P_1\ x_1\ ,\ \ldots\ P_n\ x_n$$

and

$$\underline{\text{select}}\ e_2\ \underline{\text{from}}\ P'_0\ x'_0,\ P'_1\ x'_1\ ,\ \ldots\ P'_n\ x'_n$$

are the queries that define the schema elements el_1 and el_2 respectively.

Note that since a correspondence, is a mapping it has the same semantics as a mapping. In particular, correspondence states that every instance value of schema element e_1 should also exist in the instance values of schema element e_2. Furthermore, given a source and a target schema instance, a correspondence is satisfied when the constraint described by its mapping representation is satisfied (see Definition 3.18 for mapping satisfaction).

It is a natural consequence of the definition of a correspondence that the mapping representation of a correspondence has no where clause. Recall that the **with** clause has been introduced to explicitly state the equality conditions that involve expressions from the **foreach** and the **exists** clause.

Example 4.2 *In the mapping example of Figure 3.1, the dotted arrows v_1 to v_7 represent the correspondences. The first arrow v_1, for instance, specifies that the* `name` *of the* `project` *in the source schema corresponds to* `code` *of the* `govProject`*. Formally, the specific correspondence is represented through the following mapping:*

foreach S.projects x
exists T.govProjects y
with y.govProject.code = x.project.name

■

The efficacy of using element-to-element correspondences is greatly increased by the fact that they need not be specified by a human user. They could be in fact the result of an automatic component that matches the elements of the two schemas, and then the user (data administrator) simply verifies the correctness of the results. This task is referred to in the literature as *schema matching* and has recently received considerable attention. There are a variety of methodologies that one can use to perform schema matching. Rahm and Bernstein [RB01] provide a tutorial on schema matching techniques.

A set of element correspondences may not always be one-to-one For example, we may have a correspondence from a set of of source schema elements to one target schema element. In addition, if some data value transformation is needed, then the correspondence may be annotated with the appropriate function. In the mapping representation of the correspondence, this is reflected in the **with** clause, where the right part of the equation will be, instead of an element, an arithmetic or user defined function.

Example 4.3 *Consider the case where the project* code *in the target schema of Figure 3.1 is not generated by the project* name *alone, but is a concatenation of the project* name *and the project* year*. In this case, the correspondence will be graphically represented by two lines originating from elements* name *and* year *respectively, that merge into one that ends on the element* code *of the target schema. The mapping representation of the specific correspondence will be as follows:*

foreach S.projects x
exists T.govProjects y
with y.govProject.code = *concat*(x.project.name, x.project.year)

where the function concat is used to concatenate the string values of the two elements. ■

The examples that will be used in the rest of this work will assume that a correspondence maps one source schema element to one target schema element, and the transformation function is the identity function, as in Definition 4.1. This is for presentation purposes only and does not restrict the generality, since all methods that will be presented apply equally to the case of correspondences with functions and more than one source schema element. If special consideration needs to be taken for correspondences with multiple source elements, it will be stated explicitly.

While easy to create, understand and manipulate, element correspondences are not semantically expressive enough to describe the full semantics of a transformation. As a consequence, they are inherently ambiguous. There may be many mappings that are consistent with a set of correspondences, and still, not all of them have the same semantics. A mapping generation tool needs to be able to help identify what the user (data administrator) had in mind when he/she provided a given set of correspondences, and generate plausible interpretations to produce a precise and faithful representation of the transformation, i.e., the mappings.

Example 4.4 *In the schema mapping scenario of Figure 3.1, consider correspondence* v_1 *only. One possible mapping that this correspondence alone describes is that for each project name in the first schema instance, there should be in the target schema instance a project with that name as a code. By following the same reasoning, one can make a similar interpretation of correspondence* v_2*, i.e., for every grant identifier* `gid` *in the first schema, there should be a* `funding` *in the second having the grant identifier as a funding identifier* `fid`*. But is this what a data administrator had in mind when she was providing these two arrows? It can be noticed that a grant is always related through the foreign key*

on `pi` *with a* `project`*. This means that a grant should be considered in conjunction with its associated project. So, a more natural interpretation of the two correspondences is to query the projects on the first schema, and, for each project, retrieve the grants that are for that project. Then, generate a project in the target schema and store, nested within it, the fundings of that project. This description of the transformation to take place cannot be expressed by the element correspondences alone.*

If the two correspondences are considered together, this means that projects will be associated with the mapping only when they have an associated grant (since the mapping will contain a join between projects and grants). Hence, projects that have no grants may not be associated with the mapping. Alternatively, correspondence v_1 *could also be interpreted alone (without the* v_2*). This would map all projects, even those without grants. Whether the right interpretation is a mapping that maps only projects having grants, or all the projects along with their grants (for those that have grants), is a decision that may need user involvement. Furthermore, if the* `grants` *and* `projects` *are related in multiple ways (as* `grants` *are related with* `contacts`*), then all the possible ways of associating a grant with project should be considered. Finally, it is possible that the user (data administrator) intention is to associate* `grants` *and* `projects` *in a way that is not encoded in the schema. In such a case, this information may have to be provided either directly by the user or through some other means, e.g., associations encoded in existing user defined views.*

The choice of which interpretation is the right one cannot be determined by looking at the schemas alone. A mapping generation algorithm should generate all the possible interpretations of the correspondences that are consistent with the schemas, and let the data administrator (a user) decide which of these describe the intended semantics of the correspondences. In order to do that, the first step is for the algorithm to realize how the correspondences should be grouped together. ■

Definition 4.5 (Schema Mapping Scenario) *A* Schema Mapping Scenario *is a triple*

$<\mathcal{S}, \mathcal{T}, \mathcal{V}>$ where $\mathcal{S}$ is a source schema, $\mathcal{T}$ is a target schema, and $\mathcal{V}$ is a set of correspondences between the elements of the source and the target schema.

Given a schema mapping scenario, the *schema mapping* problem is defined as the problem of finding the mappings from the first schema to the second. Our approach will be to identify mappings that are consistent with the schema structure and constraints, and the correspondences. A solution to the schema mapping problem is a *schema mapping* system.

Definition 4.6 (Schema Mapping System) *A* Schema Mapping System *is a triple $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ where $\mathcal{S}$ is a source schema, $\mathcal{T}$ is a target schema, and $\mathcal{M}$ is a set of mappings between the two schemas.*

4.2 Associations

In order to discover the intended meaning of the correspondences and generate the mappings, we first need to realize how the elements of the schemas relate to each other. The correspondences that were presented in the previous section express relationships between elements in different schemas. The next step is to find how elements within the same schema naturally relate to each other. It is not clear what exactly it means to find "natural" relationships between schema elements. We need to find a declarative definition of relationships that is independent of the particular method that is used to compute them. To do this, we look at a rather forgotten piece of database theory: the universal relation model [MUV84]. This model was developed to achieve logical access path independence in relational databases. That is, a user can view the underlying database as one relation comprised of *all* the attributes in the universe of that database. The user can then query any subset X of attributes without having to specify the join paths that must be used in the underlying database to associate the elements of X. This model defines the "natural" associations (Maier et al. [MUV84] call them "connections") among the elements of X.

The universal relation approach allows us to define, in a declarative way, independent of any computational procedure, the natural relationships between schema elements and this is what we will try to achieve.

For our work, we introduce the notion of an *association*.

Definition 4.7 (Association) *An* **association** *is a canonical query:*

select e_1, ..., e_k
from $P_0\ x_0$, $P_1\ x_1$, ... $P_n\ x_n$
where $e'_1 = e''_1$ and ... and $e'_m = e''_m$

Intuitively, an *association* represents a set of elements (of the same schema) and the relationship they may have. The set of elements represented by an association is the elements that are referred by the expressions in the select clause. Their relationship is determined by the conditions in the where clause. Many times, we will use the "*" symbol in the select clause as a short-hand of all the valid atomic type expressions that can exist in the select clause.

Example 4.8 *The query*

select *
from `S.projects` p

is a short-hand to the query

select g.`grant.gid`, g.`grant.pi`
from `S.projects` p

and is an association that represents the schema elements `name` *and* `year` *of the source schema of Figure 3.1.* ■

Since associations are queries, the notion of query dominance applies naturally to them as well.

Definition 4.9 (Union of Two Associations) *The* **union** *of two associations A and B (denoted as $A \sqcup B$) is an association with its* select *and* from *clauses consisting of the contents of the respective clauses of both A and B, and its* where *clause either contains every equality in the* where *clause of A and B, or contains expressions that logically imply every equality in the* where *clause of A and B.*

If C is a conjunction of equalities, we will abuse notation and we will use $A \sqcup C$ to denote the association A with the equalities in C appended to its where clause.

Obviously, not all associations are semantically meaningful. In database systems, there are many ways one can specify semantic relationships between schema elements. Here we consider three that are commonly used: the schema structure, the schema constraints, and the user specified associations. The next three sections describe how each one is discovered.

4.2.1 Structural Associations

One of the main ways to specify semantically meaningful relationships between the schema elements is through the schema structure, which is the structural organization (aggregation) of the schema elements. Such a set of elements can be represented by an association that will be referred to as *structural association*.

Definition 4.10 (Structural Association) *A* **structural association** *is an association of the form*

select $e_1, \ldots, e_k$
from $P_0\ x_0, P_1\ x_1\ , \ldots\ P_n\ x_n$

where expression P_0 starts with a schema root and every other expression P_i starts with the variable x_j, with $j < i$. In addition, the select *clause contains all the valid atomic type expressions that use either the schema root used by P_0 or any other variable of the* from *clause.*

Algorithm 1: Structural Association Construction

Input: A schema $\mathcal{S}$
Output: List of all structural associations $\mathcal{L}$

```
GENSTRASS(S)
(1)   L ← ∅
(2)   foreach schema root e_r of schema S
(3)     A_c ← ∅
(4)     VISIT(L, A_c, e_r)
(5)     L ← L ∪ A_c
(6)   return L

VISIT(L, A_c, e)
(1)   if [e] = Atomic
(2)     A_c ← A_c ∪ {e}
(3)   else if [e] = Rcd[l_1:τ_1, ..., l_n:τ_n]
(4)     foreach i ∈ [1, n] : i ∈ ℕ
(5)       VISIT(L, A_c, l_i:τ_i)
(6)   else if [e] = Set of τ
(7)     A_t ← A_c
(8)     VISIT(L, A_c, −:τ)
(9)     if e is mandatory
(10)      foreach A ∈ L such that A_t ⊆ A
(11)        A ← A ∪ A_c
(12)    else
(13)      L ← L ∪ {A_c}
(14)      A_c ← A_t
(15)  else if [e] = Choice[l_1:τ_1, ..., l_n:τ_n]
(16)    foreach i ∈ [1, n] : i ∈ ℕ
(17)      A_t ← A_c
(18)      VISIT(L, A_c, l_i:τ_i) L ← L ∪ {A_c}
(19)      A_c ← A_t
(20)  else
(21)    error Unknown type
(22)  return
```

Example 4.11 *Looking at the source schema of Figure 3.1, note that the elements* `gid`*,* `pi`*,* `project`*,* `budget` `recipient` *and* `private` *are all placed under the schema element* `grant`*. This means that they are all semantically related since they all specify information about a privately sponsored grant. The association that represents a group of semantically related elements, with the set being solely based on the schema structure, is a* structural association*. In particular, the association is the following:*

select g.grant.gid, g.grant.pi, g.grant.project,

```
        g.grant.budget, g.grant.recipient, r
from    S.grants g, g.grant.sponsor→private r
```

In a similar way, the structural association that describes the grants that are sponsored by the government is the following:

```
select  g.grant.gid, g.grant.pi, g.grant.project,
        g.grant.budget, g.grant.recipient, m
from    S.grants g, g.grant.sponsor→government m
```

■

For a relational schema, there is a structural association for each individual relation. For a nested schema, the structural associations are obtained by constructing a tree with a node at each set type in the schema, and with an edge between two nodes whenever the first node is a set type that contains the second (perhaps through some intermediate record types). A structural association is then the set of all atomic type elements found by using a sub-tree of that tree that has the same root, followed by a series of record projections. These steps may result in a large number of strictural associations even for small schemas. To avoid this large number, we will consider only structural associations that every expression P_i is starting with variable x_{i-1}.

For union types, the situation is similar to the case of a set type. The only difference is that a structural association needs to be created for every different choice of the union type. In addition, for a set type that is mandatory and can never be empty, instead of creating a new structural association, the atomic type elements that are reachable from the set are simply appended to the structural associations to which the parent set node belongs. Thus, structural associations of a schema denote all *vertical* data relationships that are encoded in the schema, and can exist in any instance conforming to that schema. Algorithm 1 defines the full details of the structural association generation process.

Example 4.12 *Figure 4.2 indicates the structural associations generated by applying Algorithm 1 to the two schemas of Figure 3.1. The "*" symbol has been used, as a short-*

$P_1^{\mathcal{S}}$: select * from S.projects p
$P_2^{\mathcal{S}}$: select * from S.grants g, g.grant.sponsor→private r
$P_3^{\mathcal{S}}$: select * from S.grants g, g.grant.sponsor→government m
$P_4^{\mathcal{S}}$: select * from S.contacts c
$P_5^{\mathcal{S}}$: select * from S.companies w
$P_6^{\mathcal{S}}$: select * from S.persons i

$P_1^{\mathcal{T}}$: select * from T.govProjects j
$P_2^{\mathcal{T}}$: select * from T.govProjects j, j.govProject.fundings u
$P_3^{\mathcal{T}}$: select * from T.finances f
$P_4^{\mathcal{T}}$: select * from T.companies o
$P_5^{\mathcal{T}}$: select * from T.catalog a

Figure 4.2: Source and target structural associations.

hand to indicate all the valid atomic type expressions of the select *clause. Each such structural association represents some part of the schema structure. Structural association* $P_1^{\mathcal{S}}$*, for example, represents the projects,* $P_2^{\mathcal{S}}$ *the grants with private sponsor,* $P_3^{\mathcal{S}}$ *the grants with goverment sponsor, etc. Notice that the structural associations* $P_2^{\mathcal{S}}$ *and* $P_3^{\mathcal{S}}$ *only differ in the choice of the union type element* sponsor*, since according to the algorithm definition a different structural association is generated for each choice of the union type. In addition, note that if the* fundings *element of the target schema were mandatory, i.e., its minimum cardinality were greater than zero, then the structural association* $P_1^{\mathcal{T}}$ *would not have been created, and only* $P_2^{\mathcal{T}}$ *would have been considered, since every* govProject *would have a* fundings*.* ■

4.2.2 Logical Associations

Apart from the schema structure, another way database designers may specify semantic relationships between schema elements is by using schema constraints. The constraints model additional associations between schema elements.

Example 4.13 *Every* grant *in an instance of the source schema of Figure 3.1 is related*

to a `project` *through the referential constraint from the* `project` *element of* `grant` *to the* `name` *element of* `project`*. It is natural to conclude that the elements of a* `project` *are semantically related to the elements of a* `grant`*. Thus, they can be combined together to form an association. Similarly, we can conclude that the elements of a* `grant` *are also related to the elements of a* `contact` *through the referential constraint on* `government` *and* `private`*. Hence, elements from* `private` *are naturally associated with all of* `government`*s. There are no other schema elements that can be added to the association by following that same reasoning. In that sense, the association that has just been created is maximal.*

■

Associations that are maximal like the one described in the previous example are called *logical associations*. Logical associations play an important role in the current work, since they specify possible join paths, encoded by constraints, through which two schema elements can be related. We are going to use that valuable information in order to understand whether two correspondences in a schema mapping scenario should be considered together in order to form a mapping, and also to understand how exactly they are related. This will result in mappings that are consistent with the logical[1] design of the schema.

Logical associations are computed by using the chase method [MMS79]. The chase is a classical method that has been used in query optimization [PT99], although originally it was introduced to test implications of functional dependencies. The chase was introduced for the relational model and later was extended [PT99] so that it can be applied to schemas with nested structures. Here, we extend this work even further so that it can be applied to structures that contain union (choice) types.

A chase procedure is a sequence of chase steps. A chase step is applied to an association by using a schema constraint. Although, the chase was defined to work for tuple and equality generating dependences, here we will consider an NRIs, and an extended version

[1]That is why they are referred to as *logical associations*.

of the chase that works with nested dependencies like the NRIs [Pop00]. The output of a chase step is a new association if the NRI constraint can be applied or the same association if it cannot. An NRI constraint can be applied to an association if its **foreach** part is dominated by the association. (Recall that both an association and the **foreach** part of a constraint are queries.) If the NRI constraint can be applied, its **exists** part is appended to the association. Appended means that every variable binding is appended to the from clause of the association, and every condition in the **with** part of the where clause of the **exists** part is also appended to the where clause of the association if it is not already there (with a possible renaming of variables to avoid any conflicts).

Definition 4.14 (Chase Step) *Given a schema constraint:* **foreach** F **exists** R **with** W *and an association* A*:* select * from F_A where C_A*, a* **chase step** *can be applied to the association* A *if and only if both the following conditions are met:*

1. $F \dot{\preceq} A$ *and*
2. *association* select * from F_{select}, R_{select} where F_{where} and R_{where} and W *is not dominated by association* A *(where* F_{select}*,* R_{select}*,* F_{where} *and* R_{where} *are the* select *and* where *clauses of* F *and* R *respectively).*

In such a case, the chase step is a rewriting of A *to:*

select * from F_A, R_{from} where C_A and R_{where} and W

Thus, whenever the **foreach** part of an NRI matches a sub-part of an association A, the chase can be applied and adds to A (if it is not there already) the **exists** part of the constraint together with the condition in its **with** clause.

In general, more than one chase step can be applied to an association. Given an association A and a set Σ of schema constraints, we denote by $\texttt{Chase}_\Sigma(T)$ the final association A' produced by a sequence of chase steps starting at A and using constraints

from Σ. By final association, we mean that no chase steps are applicable to A'. Maier et al. [MMS79] have shown that for the relational model, two different chase sequences with the same set of dependencies generate identical results. Popa [Pop00] has shown a similar result for the case of the nested relational model. Hence, the result $\texttt{Chase}_\Sigma(T)$ is unique. Of particular interest to us is the chase applied to structural associations, which is used to compute logical relationships that exist in the schema. A logical association can then be formally defined as the result of the chase.

Definition 4.15 (Logical Association) *A* **logical association** *is the result of chasing a structural association A with the set of schema constraints Σ (denoted as $\texttt{Chase}_\Sigma(A)$) of the schema, until no further chase step is possible.*

Example 4.16 *Consider the structural association $P_2^{\mathcal{S}}$ of the Figure 4.2. Constraint*

f_4: **foreach** S.companies c
exists S.persons s
with s.person.SSN=c.company.CEO

cannot be applied to $P_2^{\mathcal{S}}$ since there is no renaming[2] *of the variable c so that the binding on* S.companies *exists in the* from *clause of $P_2^{\mathcal{S}}$. Similarly for constraint f_5. However, constraint f_1:*

foreach S.grants g
exists S.projects p
with p.project.name=g.grant.project

can be applied. The chase step will rewrite association $P_2^{\mathcal{S}}$ to:

select *
from S.grants g, g.grant.sponsor→private r, S.projects p
where p.projects.name=g.grant.project

Applying constraint f_2:

[2]The renaming is needed to check for dominance.

foreach S.grants g, g.grant.sponsor→private r
exists S.contacts c
with c.contact.cid=r

will rewrite the association to:

select *
from S.grants g, g.grant.sponsor→private r,
S.projects p, S.contacts c
where p.project.name=g.grant.project and c.contact.cid=r

No other constraint can be applied to this last association, hence it is a logical association. Figure 4.3 indicates the logical associations for the two schemas of Figure 3.1. Note that, in some cases, parts of the schema may appear more than once. For example, association A_5 *has two bindings on* persons. *The first is generated by applying constraint* f_4 *to the structural association* $P_5^{\mathcal{S}}$ *while the second by applying constraint* f_5 *to the result of the application of* f_4. ■

4.2.3 User Associations

The structural and logical associations encode semantic relationships between the schema elements that data administrators have encoded in the schema during schema design through the structuring of the elements and by specifying appropriate schema constraints. However, this is not the only kind of relationship that can exist between schema elements, but rather those that one can identify by simply looking at the schema structure and constraints.

Another valuable source of information is any existing queries or views that a user has written. If, for example, the user performs a query that joins two schema elements in a specific way, it means that these elements are related in the way the query specifies. Associations that can be inferred by using user queries, views, or other input from the user, are referred to as *user associations*.

$A_1^{\mathcal{S}}$: select *
from S.projects p

$A_2^{\mathcal{S}}$: select *
from S.grants g,
g.grant.sponsor→private r,
S.projects p, S.contacts c
where p.project.name=g.grant.project
and c.contact.cid=r

$A_3^{\mathcal{S}}$: select *
from S.grants g,
g.grant.sponsor→government m,
S.projects p, S.contacts c
where p.project.name=g.grant.project
and c.contact.cid=m

$A_4^{\mathcal{S}}$: select *
from S.contacts c

$A_5^{\mathcal{S}}$: select *
from S.companies w, S.persons n,
S.persons s,
where s.person.SSN=o.company.CEO and
n.person.SSN=o.company.owner

$A_6^{\mathcal{S}}$: select *
from S.persons s

$A_1^{\mathcal{T}}$: select *
from S.govProjects j

$A_2^{\mathcal{T}}$: select *
from S.govProjects j,
j.govProject.fundings u,
S.finances f, S.companies o
where j.funding.finId=f.finances.finId
and
f.finances.company=o.company.cname

$A_3^{\mathcal{T}}$: select *
from S.finances f, S.companies o
where f.finances.company=o.company.cname

$A_4^{\mathcal{T}}$: select *
from S.companies o

$A_5^{\mathcal{T}}$: select *
from S.catalog a

Figure 4.3: Source and target logical associations.

Definition 4.17 (User Association) *A* **user association** *is an association that is specified by a user query or mapping.*

Example 4.18 *Consider the case where a data administrator has manually defined the following mapping between the two schemas of Figure 3.1:*

foreach S.contacts c, S.persons i
where i.person.SSN=c.contact.cid
exists T.catalog a
where a.entry.phone=c.contact.phone and
a.entry.name=i.person.name

This mapping specifies that it makes sense for a contact to be associated with person information if the contact identifier cid *is equal to the* SSN *of the person. Since the data*

administrator provides this association through the mapping, one can use it along with the structural and logical associations of the schema. The following user association can be derived from this user mapping.

```
select  *
from    S.contacts c, S.persons i
where   i.person.SSN=c.contact.cid
```

Taking into account user associations, the definition of a logical association can be modified to:

Definition 4.19 (Logical Association - Revisited) *A* **logical association** *is the result of chasing a structural or a user association A with the set of schema constraints Σ (denoted as $\texttt{Chase}_\Sigma(A)$) of the schema, until no further chase step is possible.*

■

4.3 Semantic Translation

4.3.1 Coverage

To perform schema mapping, we seek to interpret correspondences in a way that is consistent with the semantics of both the source and target schemas. To do that, we use logical associations. We call this interpretation process *semantic translation*. Since we use semantics encoded in the schema structure and constraints, we call the mappings generated as a result of the semantic translation *semantically complete mappings*. In this section, we formalize this intuition and give an algorithm for interpreting correspondences. This corresponds to the box labeled "Mapping Generation" of Figure 4.1.

Example 4.20 *Consider the correspondences v_1 and v_2 in the schema mapping scenario of Figure 3.1. Recall that correspondences are trivial forms of mapping. Hence, the correspondences v_1 and v_2 are each mappings:*

v_1: **foreach** S.projects x
exists T.govProjects y
with y.govProject.code=x.project.name

v_2: **foreach** S.grants z
exists T.govProjects y, y.fundings w
with w.funding.fid=z.grant.gid

The correspondence v_1 is between two elements that belong to the structural associations $P_1^{\mathcal{S}}$ and $P_2^{\mathcal{T}}$ in the source and target schema, respectively (Figure 4.2). Similarly, v_2 is between two elements that belong to the structural associations $P_2^{\mathcal{S}}$ and $P_2^{\mathcal{T}}$ of the source and target schema, respectively. The interpretation that considers these two correspondences as mappings, independently of one another, ignores the semantics of the source and target schemas. It indicates that fundings *are created from* grants*, but does not say that* fundings *must be nested within a* govProject *generated from the appropriate project in the source (that is, the project receiving the grant). In particular, these two correspondences can be combined together to form the following mapping:*

v_{12}: **foreach** S.projects p, S.grants g
where p.project.name=g.grant.project
exists T.govProjects j, j.fundings u
with j.govProject.code=p.project.name and
u.funding.fid=g.grant.gid

If the correspondence v_3 is also considered, the above mapping will be extended with the condition u.funding.pi=g.grant.pi in the **with** *clause. If correspondences v_4 and v_5 are considered as well, the* **foreach** *and* **exists** *clauses of the mappings will also be extended. Considering also correspondence v_6 together with v_1, v_2, v_3, v_4 and v_5, may not be so natural, since the element* cname *on which v_6 is defined is not semantically related to the source schema elements of the correspondences v_1 to v_5.* ■

We consider next an algorithm for systematically performing the reasoning of the above example. We use the logical associations since they naturally provide groups of elements that are semantically related. In particular, to determine sets of correspondences

that can be interpreted together, we find sets of correspondences that all use elements in one source logical association and target elements in one target logical association. As seen in the previous section, logical associations are not necessarily disjoint. For example, $A_2^{\mathcal{S}}$ and $A_3^{\mathcal{S}}$ of Figure 4.3 both include `project` information; however, $A_2^{\mathcal{S}}$ also includes information about the `grants` having a `private sponsor` that are related to the project, along with information about the sponsor, while $A_3^{\mathcal{S}}$ describes the same information for grants having government sponsors. Thus, a correspondence can be relevant to several logical associations (in both source and target). Rather than looking at each individual correspondence, the mapping algorithm looks at each pair of source and target logical associations. For each such pair, it computes one or more mappings that are consistent with the the correspondences. These mappings are between elements of these two logical associations and express how the information described by the source logical association is mapped into the information described by the target logical relation. The generated mappings are inter-schema constraints, in particular, inter-schema NRIs.

Given a pair of source and target logical associations, deciding whether a correspondence v should be considered or not, i.e., finding whether the schema elements it relates belong in the two associations, is slightly more complicated than what we have described above. It is not enough to check whether the logical association includes the element names involved in correspondence v. This is ambiguous, in general, since even elements within the same schema may have the same name. Moreover, the *same* element of a schema may be included more than once in a logical association (Example 4.30 below). To precisely identify whether an element in a logical association is an element used by a correspondence, one needs to match the variables in the correspondence with the variables of the logical association. Specifically, one needs to find a *renaming function* from the variables of a correspondence into the variables of a logical relation. Since nested schemas are used, there is the additional issue of matching paths in the schema, rather than flat element references. To formally define this, the notion of *coverage* is intro-

duced. Note that this is a similar problem to that encountered in answering queries using views [LMSS95].

Definition 4.21 (Coverage) *A correspondence*

$$\begin{array}{lll} v\text{: } \underline{\textbf{foreach}} & P_0\ x_0,\ P_1\ x_1\ ,\ \ldots\ P_n\ x_n \\ \quad \underline{\textbf{exists}} & P'_0\ x'_0,\ P'_1\ x'_1\ ,\ \ldots\ P'_n\ x'_n \\ \quad \underline{\textbf{with}} & e' = e \end{array}$$

is **covered** *by the pair of associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ if there exist two renaming functions: h from the variables in the* **foreach** *clause of the correspondence into variables of $A^{\mathcal{S}}$, and h' from the variables in v's* **exists** *clause into variables of $A^{\mathcal{T}}$ such that, for every variable binding $P_i\ x_i$ in the* **foreach** *clause of v, variable binding $h(P_i)\ h(x_i)$ is in the* from *clause of $A^{\mathcal{S}}$ and, similarly, for every variable binding $P'_i\ x'_i$ in the* **exists** *clause of v, variable binding $h'(P'_i)\ h'(x'_i)$ is in the* from *clause of $A^{\mathcal{T}}$. The pair $<h, h'>$ is referred to as the* way of coverage*, or simply* coverage*. We may also say that a correspondence is covered* through *the renaming functions $<h, h'>$. The* result *of a coverage $<h, h'>$ of correspondence v, noted as $r^{v,<h,h'>}$, is the expression $h'(e')$=$h(e)$.*

The above definition states that a correspondence v: **foreach** P_S **exists** P_T **with** D **is covered** by the pair of associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ if $P_S \dot{\preceq} A^{\mathcal{S}}$ and $P_T \dot{\preceq} A^{\mathcal{T}}$.

Definition 4.22 *A correspondence v:* **foreach** P_S **exists** P_T **with** $e' = e$ *is* **covered** *by mapping m:* **foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D *if v is covered by the pair of associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$, through the renaming functions $<h, h'>$ and equality $h'(e')$=$h(e)$ is in D.*

Given a mapping m between two schemas it is always possible to extract the correspondences that are covered by that mapping.

Example 4.23 *Correspondence*

v_1: **foreach** S.projects p
exists T.govProjects j
with j.govProject.code=p.project.name

is covered by mapping

m_{cov}: **foreach** S.grants g, g.grant.sponsor→government r, S.projects p, S.contacts c
where p.project.name=g.grant.project
and c.contact.cid=r
exists T.govProjects j, j.govProject.fundings u, T.finances f, T.companies o
where j.funding.finId=f.finances.finId and
f.finances.company=o.company.cname
with j.govProject.code = p.project.name and u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and f.finances.amount=g.grant.budget and
f.finances.sponsorPhone=c.contact.phone

since there are two homomorphisms (the identity homomorphisms) from the **foreach** *and* **exists** *parts of the correspondence to the* **foreach** *and* **exists** *associations of mapping* m_{cov}*, and the result of applying these homomorphisms on the* **with** *clause of correspondence* v_1*, exists in the* **with** *clause of mapping* m_{cov}*.*

On the other hand, given the mapping m_{cov}*, five covered correspondences can identified and extracted. Considering the last equality of the* **with** *clause, for example, the extracted correspondence is*

v_{ext}: **foreach** S.contacts c
exists S.finances f
with f.finances.sponsorPhone=c.contact.phone

■

To test whether a schema element plays any role in a query, constraint or mapping, we need to check whether there is a renaming function from the variables of the element query to the variables of the query, constraint or mapping, respectively, and whether the renaming of its select clause expression is used in any of the expressions of the constraint. Recall that associations are represented as queries, mappings are constraints, and correspondences are special forms of mappings.

Definition 4.24 (Element Participation) *A schema element e* **participates (or used)** *in a query q if there is a renaming function h from the variables of the element query of e to the variables of the query q such that the renamed expression exp in the element query of e is used in any of the expressions in the* from *or* where *clause of the query q. The element is also said to participates in a constraint (or a mapping, since a mapping is a constraint) F*: **foreach** *X* **exists** *Y* **with** *C if it participates in association* $X \sqcup Y \sqcup C$.

4.3.2 Mapping Generation Algorithm

Our goal is to generate mappings that are consistent with the the correspondences and the constraints of the schemas. This section shows how the logical associations and the notion of coverage are used to generate the logical mappings.

Definition 4.25 (Constraint Satisfaction by the Mappings) *A mapping is consistent with a correspondence (or a constraints) if the correspondence (or the constraint) is logically implied by the mappings. Alternatively, we may also say that the mapping respects or does not violate the correspondence or the schema constraint.*

We will consider a special class of mappings that have this specific property. Thse mappings are referred to as *semantically complete mappings*.

Definition 4.26 (Semantically Complete Mapping) *Let* $\mathcal{S}$ *and* $\mathcal{T}$ *be a pair of source and target schemas and* $\mathcal{C}$ *a set of correspondences between them. A* **semantically complete mapping** *is a mapping* **foreach** A^S **exists** A^T **with** D, *where* A^S *and* A^T *are logical associations in the source and the target schemas, respectively, and D is the conjunction of the* **with** *clause of each correspondences in C that is* covered *by the pair* $<A^S, A^T>$ *(provided that at least one such correspondence exists), after a respective renaming functions has been applied to each one of them, such that each schema element is used by at most one of the covered correspondences.*

Definition 4.27 (Mapping Universe) *Given a source and a target schema, $\mathcal{S}$ and $\mathcal{T}$, along with a set $\mathcal{C}$ of correspondences from $\mathcal{S}$ to $\mathcal{T}$, the* **mapping universe** $\mathcal{U}^{\mathcal{C}}_{\mathcal{S},\mathcal{T}}$ *is the set of all the semantically complete mappings.*

This section presents an algorithm that, given two schemas $\mathcal{S}$ and $\mathcal{T}$ and a set of correspondences $\mathcal{C}$ from the first to the second, produces all the mappings of the mapping universe $\mathcal{U}^{\mathcal{C}}_{\mathcal{S},\mathcal{T}}$. The algorithm generates the semantically complete mappings as follows. For a given pair of source and target logical associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$, mapping

$$
\begin{array}{lll}
m_{tmp}\text{: } \textbf{foreach} & F_{A^{\mathcal{S}}} \\
\quad \text{where} & W_{A^{\mathcal{S}}} \\
\textbf{exists} & F_{A^{\mathcal{T}}} \\
\quad \text{where} & W_{A^{\mathcal{T}}} \\
\textbf{with} &
\end{array}
$$

is initially constructed, where $F_{A^{\mathcal{S}}}$, $W_{A^{\mathcal{S}}}$, $F_{A^{\mathcal{T}}}$ and $W_{A^{\mathcal{T}}}$ are the from and where clauses of the associations $A^{\mathcal{S}}$ and $A^{\mathcal{T}}$, respectively. Note that the **with** clause is initially empty. Then, for every correspondence $v \in \mathcal{C}$, the algorithm checks whether it is covered or not by the pair of associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$. If it is covered, the result of the coverage is appended to the **with** clause of the mapping. The final mapping is a semantically complete mapping of the mapping universe $\mathcal{U}^{\mathcal{C}}_{\mathcal{S},\mathcal{T}}$. Note, also, that since the order of the equalities in the **with** clause plays no role, the generated mapping is independent of the order in which the correspondences are checked for coverage.

Example 4.28 *Considering the pair of source and target logical associations $A^{\mathcal{S}}_3$ and $A^{\mathcal{T}}_2$ in Figure 4.3 the following mapping is constructed, with an empty* **with** *clause:*

$$
\begin{array}{lll}
m_{tmp}\text{: } \textbf{foreach} & \texttt{S.grants } g, \ g\texttt{.grant.sponsor} \rightarrow \texttt{government } r, \ \texttt{S.projects } p, \ \texttt{S.contacts } c \\
\quad \text{where} & p\texttt{.project.name} = g\texttt{.grant.project} \\
& \text{and } c\texttt{.contact.cid} = r \\
\textbf{exists} & \texttt{T.govProjects } j, \ j\texttt{.govProject.fundings } u, \ \texttt{T.finances } f, \ \texttt{T.companies } o \\
\quad \text{where} & j\texttt{.funding.finId} = f\texttt{.finances.finId and} \\
& f\texttt{.finances.company} = o\texttt{.company.cname} \\
\textbf{with} &
\end{array}
$$

Consider the value correspondence:

v_1: **foreach** S.projects x
exists T.govProjects y
with y.govProject.code = x.project.name

representing the dotted arrow v_1 in Figure 3.1.

If variable x of its **foreach** *clause is renamed to variable p, then the bindings of its* **foreach** *clause (in this case there is only one) are included in the set of bindings in the* **foreach** *clause of mapping m_{tmp}. Similarly, if variable y of v's* **exists** *clause is renamed to variable j, then the bindings of its* **exists** *clause (in this case there is also only one) are included in the set of bindings in the* **exists** *clause of mapping m_{tmp} below. Let $h_f : \{x \rightarrow p\}$ be the renaming function for the variables in the* **foreach** *part, and $h_c : \{y \rightarrow j\}$ be the renaming function for the variables in the* **exists** *part, the binary predicate $h_c(e_c) = h_f(e_f)$ is constructed for each binary predicate $e_c = e_f$ in the* **with** *clause of the value correspondence and appended to the with clause of the mapping m_{tmp}. This means that mapping m_{tmp} becomes:*

m_{tmp}: **foreach** S.grants g, g.grant.sponsor→government r, S.projects p, S.contacts c
where p.project.name=g.grant.project and
c.contact.cid=r
exists T.govProjects j, j.govProject.fundings u, T.finances f, T.companies o
where j.funding.finId=f.finances.finId and
f.finances.company=o.company.cname
with j.govProject.code = p.project.name

Following similar steps for the remaining correspondences, the final mapping is:

m_{final}: **foreach** S.grants g, g.grant.sponsor→government r, S.projects p, S.contacts c
where p.project.name=g.grant.project and
c.contact.cid=r
exists T.govProjects j, j.govProject.fundings u, T.finances f, T.companies o
where j.funding.finId=f.finances.finId and
f.finances.company=o.company.cname

with j.govProject.code = p.project.name and u.funding.fid=g.grant.gid and u.funding.pi=g.grant.pi and f.finances.amount=g.grant.budget and f.finances.sponsorPhone=c.contact.phone

Note that correspondences v_6 and v_7 are not covered by the pair of these two logical associations. For example, for the correspondence v_6:

foreach S.companies z
exists T.companies i
with i.company.cname = z.company.cname

*there is no renaming function f to map variable z so that the binding "*S.companies $h_f(z)$*" is included in the* **foreach** *clause of association $A_2^{\mathcal{S}}$.* ■

Assuming that the source schema has N logical associations and the target schema has M, there will be $N \times M$ pairs. This may result in the generation of a large number of mappings. In order to reduce the number of generated mappings, we use the following heuristic. If a mapping m is generated by the coverage of a number of correspondences by the pair of associations $<A^{\mathcal{S}}, A^{\mathcal{T}}>$, the pair of associations $<B^{\mathcal{S}}, A^{\mathcal{T}}>$ (or $<A^{\mathcal{S}}, B^{\mathcal{T}}>$) is not considered if $B^{\mathcal{S}} \dot{\preceq} A^{\mathcal{S}}$ (or $B^{\mathcal{T}} \dot{\preceq} A^{\mathcal{T}}$) and no additional correspondences are covered by pair $<B^{\mathcal{S}}, A^{\mathcal{T}}>$ (or $<A^{\mathcal{S}}, B^{\mathcal{T}}>$) compared to the correspondences covered by $<A^{\mathcal{S}}, A^{\mathcal{T}}>$. In the general case, for the same set of correspondences, the mapping generated by pair $<B^{\mathcal{S}}, A^{\mathcal{T}}>$ (or $<A^{\mathcal{S}}, B^{\mathcal{T}}>$) is not equivalent to the mapping generated by $<A^{\mathcal{S}}, A^{\mathcal{T}}>$. However, due to the way mappings are evaluated and the target instance produced, the data generated by the mapping based on pair $<B^{\mathcal{S}}, A^{\mathcal{T}}>$ (or $<A^{\mathcal{S}}, B^{\mathcal{T}}>$) is included in the data generated by the mapping which is based on pair $<A^{\mathcal{S}}, A^{\mathcal{T}}>$. Hence, this optimization heuristic can be applied.

Example 4.29 *Consider the schema mapping scenario of Figure 3.1, but with only one correspondence, that is v_5, the target schema logical association $A_3^{\mathcal{T}}$ from Figure 4.3, and the source schema logical association $A_4^{\mathcal{S}}$ from the same figure. The pair of associations $<A_4^{\mathcal{S}}, A_3^{\mathcal{T}}>$ covers correspondence v_5 and generates the mapping*

m: **foreach** S.contacts c
exists S.finances f, S.companies o
where f.finances.company=o.company.cname
with f.finances.sponsorPhone=c.contact.phone

which states that each phone number of a contact *information from the source must appear as* sponsorPhone *of* finances *in the target. If, instead of association* $A_4^{\mathcal{S}}$*, association* $A_3^{\mathcal{S}}$ *is considered, the mapping generated is the mapping:*

m': **foreach** S.contacts c, S.grants g, g.grant.sponsor→government r
where c.contact.cid=r
exists T.finances f, T.companies o
where f.finances.company=o.company.cname
with f.finances.sponsorPhone=c.contact.phone

which, states that each phone number of a contact *information of a government sponsored grant from the source must appear as* sponsorPhone *of* finances *in the target. The constraint expressed by mapping* m *includes the constraint expressed by mapping* m'*. However, our heuristic takes the more general solution since, due to the lack of any correspondence in* grants*, the algorithm has no reason to consider* grants *in the generated mapping.*

Figure 4.4 illustrates the same principle for the target schema. In the first case, since there is no correspondence to the fundings *element, the associations that include both* govProjects *and* fundings *will not be considered, but only the one that includes* govProjects*. In the second case, since there is no correspondence on* govProjects*, only the association that has* finances *and* companies *will be considered. Note that* finances *alone is not considered since it is not a logical association. The reason is that the second foreign key indicates that for every* finances *element there is always a related* company*.*

■

Given a pair of source and target logical associations and a correspondence v, there may be many ways the correspondence can be covered. In particular, there may be many

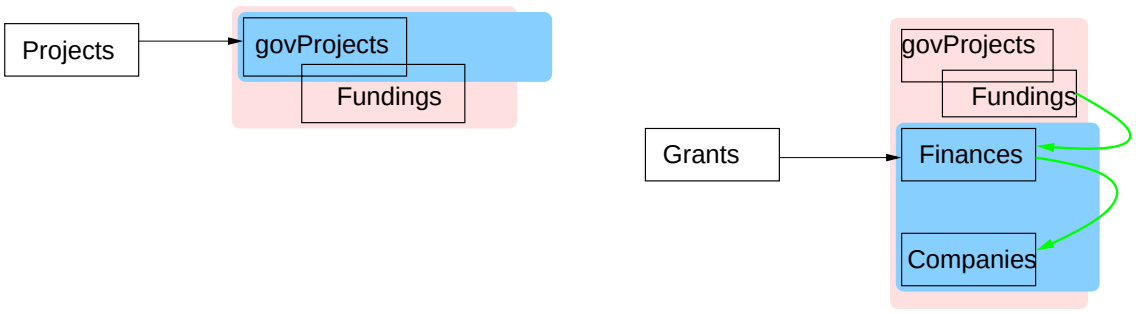


Figure 4.4: Illustration of the optimization heuristic.

renaming functions to map the variables. Each way of coverage captures some possible semantic interpretation of the correspondence. Hence, all the mappings generated by the different ways of coverage should be considered.

Example 4.30 *Consider the pair of associations* $<A_5^{\mathcal{S}}, A_4^{\mathcal{T}}>$ *from Figure 4.3 and the correspondence:*

v_6: **foreach** S.companies z
exists T.companies i
with i.company.cname = z.company.cname

There is only one way v_6 *can be covered by this specific pair of associations, and this is by the two renaming functions* $h_f : \{z \rightarrow w\}$ *and* $h_c : \{i \rightarrow o\}$*. On the other hand, the correspondence*

v_7: **foreach** S.persons d
exists T.companies i
with i.company.leader = d.person.name

can be covered by this pair of associations in two ways. One is $h_f : \{d \rightarrow n\}$ *and* $h_c : \{i \rightarrow o\}$ *and the second is* $h'_f : \{d \rightarrow s\}$ *and* $h'_c : \{i \rightarrow o\}$*. The first way of coverage, combined with the coverage for correspondence* v_6 *gives the mapping:*

m_1: **foreach** S.companies w, S.persons n, S.persons s
where s.person.SSN=o.company.CEO and
n.person.SSN=o.company.owner

exists T.companies o
with o.company.cname=w.company.cname and
o.company.leader = n.person.name

while the second gives

m_2: **foreach** S.companies o, S.persons n, S.persons s
where s.person.SSN=o.company.CEO and
n.person.SSN=o.company.owner
exists T.companies o
with o.company.cname=o.company.cname and
o.company.leader = s.person.name

The difference between the first and the second mapping is how the name of the person is interpreted. In the first case, the name of the person is considered to be the name of the CEO of the company, while in the second case, the person is the owner of the company. Both are valid interpretations of the correspondences v_6 and v_7. Without any other extra information, no decision can be made by the algorithm as to which of the two is preferable. ■

The described procedure is summarized in Algorithm 2.

Note that the mappings in the mapping universe are guaranteed, by construction, not to violate the constraints of the target schema.

Example 4.31 *Note that the* **with** *clause of mapping m_{final} in Example 4.28 contains no equality specifying the value of the elements* leader *or* cname *of* company*. However, the* **exists** *clause of the mapping contains a variable binding to* companies *and a condition in its* where *clause that involves it. The reason these statements were generated by the algorithm is to guarantee that every value that may appear as a* company *in* finances*, also appears as a* cname *of a* company*. That way, the consistency of the schema is preserved, since constraint f_7 is not violated.* ■

Of course, the set of mappings generated by the mapping generation algorithm is not the set of all mappings that can exist between the source schema and the target schema.

Algorithm 2: Logical Mappings Generation

Input: A source schema $\mathcal{S}$
A target schema $\mathcal{T}$
A set of value correspondences $\mathcal{V}$
Output: The set of all the semantically complete mappings $\mathcal{M}$

GENERATELOGICALMAPPINGS($\mathcal{S}$, $\mathcal{T}$, $\mathcal{V}$)
(1) $\mathcal{M} \leftarrow \emptyset$
(2) $\mathcal{A}^{\mathcal{S}} \leftarrow$ Logical Associations of $\mathcal{S}$
(3) $\mathcal{A}^{\mathcal{T}} \leftarrow$ Logical Associations of $\mathcal{T}$
(4) **foreach** pair $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ of $\mathcal{A}^{\mathcal{S}} \times \mathcal{A}^{\mathcal{T}}$
(5) $V=\{v \mid v \in \mathcal{V} \wedge v$ is covered by $<A^{\mathcal{S}}, A^{\mathcal{T}}>\}$
(6) *// If no correspondences are covered*
(7) **if** $V = \emptyset$
(8) continue;
(9) *// Optimization*
(10) **if** $\exists <X,Y> = <B^{\mathcal{S}}, A^{\mathcal{T}}>$ with $B^{\mathcal{S}} \dot{\preceq} A^{\mathcal{S}}$ or
(11) $<X,Y> = <A^{\mathcal{S}}, B^{\mathcal{T}}>$ with $B^{\mathcal{T}} \dot{\preceq} A^{\mathcal{T}}$
(12) $V'=\{v \mid v \in \mathcal{V} \wedge v$ is covered by $<X, Y>\}$
(13) **if** $V' = V$
(14) continue;
(15) $\delta_v = \{<h^s, h^t> | v \in V \wedge v$ is covered using renamings $<h^s, h^t>\}$
(16) $W_v =$ the **with** clause of correspondence v
(17) *// For every combination of coverages of the correspondences*
(18) **foreach** $<\delta_{v_1}^{h_1^s,h_1^t}, \delta_{v_2}^{h_2^s,h_2^t}, \ldots \delta_{v_n}^{h_n^s,h_n^t}>$ of the $v_i \in V$
(19) $W = \emptyset$
(20) **foreach** $h_i^s(e_i)=h_i^t(e_i') : v_i \in V \wedge <h_i^s, h_i^t> \in \delta_{v_i} \wedge W_{v_i}=\{e_i=e_i'\}$
(21) **if** $h(e_i)$ and $h(e_i')$ are not used in W
(22) $W = W \cup \{h_i^s(e_i)=h_i^t(e_i')\}$
(23) $M \leftarrow$ **foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** W
(24) $\mathcal{M} \leftarrow \mathcal{M} \cup \{M\}$
(25) **return** $\mathcal{M}$

However, it is the set of all mappings that are consistent with the schema structure and constraints and the given correspondences. The reason we consider this kind of mappings is because it is generally believed that the schema structure and the schema constraints are the natural ways through which the database designers describe the semantics of the data [MU83].

4.4 Data Translation

The mappings generated during the semantic translation process specify how the data of the two schemas relate to each other. In the case of a data exchange system, the next step is the *data translation* process. Recall that in a data exchange system, data structured under the first schema needs to be restructured and translated to an instance of the second schema. This means that we need to have mappings that fully specify a target schema instance based on a given source schema instance. The generation of such complete mappings is referred to as the *data exchange*. Data exchange is the topic of this section, and the procedure described corresponds to the boce labeled "Data Translation" in Figure 4.1.

According to Fagin et al. [FKMP03b] the data exchange problem is defined as follows:

Definition 4.32 (Data Exchange Problem) *Given a schema mapping system* $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ *and an instance* $\mathcal{I}$ *of the Schema* $\mathcal{S}$*, the* data exchange *or* data translation *problem is defined as the problem of finding an instance* $\mathcal{I}_t$ *of schema* $\mathcal{T}$ *that satisfies the mappings* $\mathcal{M}$*. Such an instance is called a* solution *to the data exchange problem.*

This section describes an algorithm for finding a *solution* to a data-exchange problem.

The mappings generated by Algorithm 2 are tuple generating dependencies (in the relational sense) between source logical associations and target logical associations (which are relational views of their respective schemas). For example, v_{12} of Example 4.20 is an inclusion dependency $A_2^S[\texttt{name},\texttt{gid}] \subseteq A_2^T[\texttt{code},\texttt{fid}]$ meaning that the projection of logical association A_2^S on `name` and `gid` must be contained in the projection of A_2^T on `code` and `fid`. If we also take the correspondences v_3 and v_4 into account, then v_{12} is replaced by a similar inclusion dependency, $A_2^S[\texttt{name},\texttt{gid},\texttt{pi},\texttt{budget}] \subseteq A_2^T[\texttt{code},\texttt{fid},\texttt{pi},\texttt{amount}]$, that "specifies" two more atomic elements of the target. Still, not all atomic elements of the target logical relation may be specified by the mapping. In order to materialize the target, we must fill in the values for the undetermined atomic elements. Null values

may not always be sufficient. Moreover, we do not want to materialize a target logical relation, but rather a nested instance over which the logical association is only a flat view. For example, null values are not what we want for the element `finId`. Even though A_2^T gathers together all the elements of the logical relation, in the nested schema, the elements are partitioned into three different parts of the schema. In order to implement this partitioning without loss of information, `finId` will have to play the role of a placeholder to be created in two parts of the schema.

These issues are addressed by the second phase of our translation process: **data translation**. Specifically, two main problems are considered: 1) *creation of new values in the target*, whenever such values are needed and whenever they are not specified by the dependencies, and 2) *grouping of nested elements*, a form of data merging that is also not specified by the dependencies but is often desirable. We illustrate our solution in the following subsections. In particular, we first show how we can make sure that the values we will generate in the target instance will respect the target schema constraints, how we can achieve the right groupings when populating the target schema with values, and finally, how the values will be generated. The output of that process is a set of rules that fully specify the target instance, and is the output of box labeled "Data Translation" in Figure 4.1. These rules have to be transformed into actual queries in the native query language of the source database so that can be executed. This transformation process (box labeled "Query Generation" in Figure 4.1) is described in Section 4.4.5.

4.4.1 Creation of New Values in the Target.

Consider the simple mapping scenario of Figure 4.5(a). The source ($\texttt{Emps}_1$) and the target (`Emps`) are sets of employee (`Emp`) elements. An `Emp` element in the source has atomic subelements `A`, `B` and `C`, while an `Emp` element in the target has an extra atomic subelement `E`. For the purpose of this discussion, we choose to use the abstract names `A`, `B`, `C` and `E` because we will associate several scenarios with these elements. In the

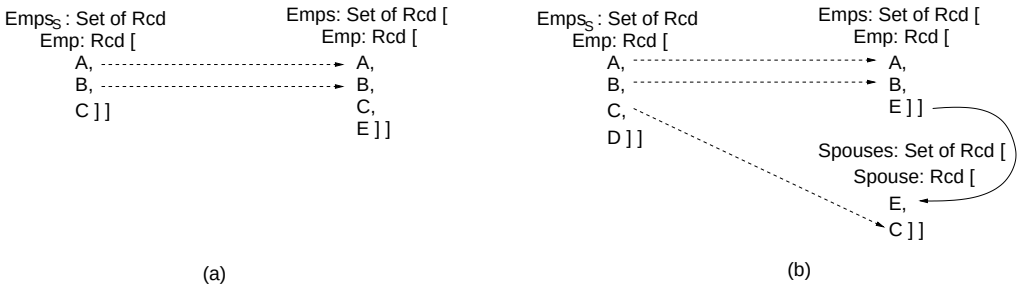


Figure 4.5: Creation of new values in the target.

mapping, the two source elements `A` and `B` are mapped into the target elements `A` and `B`, while `C` and `E` in the target are left unmapped. The mapping:

foreach $\texttt{Emps}_s\ x$
exists $\texttt{Emps}\ y$
with $y.\texttt{Emp.A}=x.\texttt{Emp.A}$ and $y.\texttt{Emp.B}=x.\texttt{Emp.B}$

does not specify any value for either `C` or `E`. However, to populate the target we need to decide what values (if any) to use for these elements. A frequent case in practice is the one in which an unmapped element does not play a crucial role for the integrity of the target. For example, `A` and `B` could be employee name and salary, while `C` and `E` could be address and, date of birth, respectively. Creating a null value for either `C` or `E` is then sufficient. Or, if the unmapped element is optional in the target schema, then we can leave it out entirely. For example, if `C` is optional while `E` is not optional but nullable, we translate the mapping to the following rule:

foreach $\texttt{Emps}_S\ x$
let $a=x.\texttt{Emp.A}$, $b=x.\texttt{Emp.B}$
exists $\texttt{Emps} \leftarrow \langle\ \texttt{Emp} = \langle\ \texttt{A} = \text{a}, \texttt{B} = \text{b}, \texttt{E} = \text{null}\ \rangle\ \rangle$

The **exists** clause of the rule explicitly describes the structure of the data that is to be constructed. This is in accordance with the syntax followed by the return clause of query languages like XQuery. It specifies that a tuple value `Emp` is created and appended to the set of values `Emps`.

The element E could be a key in the target relation, e.g., E could be employee id. The intention of the mapping would be, in this case, to copy employee data from the source, and assign a new id for each employee in the target. Thus, a non-null value for E is needed for the integrity of the target.

A target element E *is* **needed** *if* E *is (part of) a key or foreign key or is both not nullable and not optional.*

For our example, we create a *different* but *unique* value for E, for *each* combination of the source values for A and B. Technically speaking, values for E are created by using a one-to-one (Skolem) function $f_{\mathtt{E}}[\mathtt{A},\mathtt{B}]$. The rule that translates the mapping in this case is the same as the previous rule except that we use $f_{\mathtt{E}}[\mathtt{A},\mathtt{B}]$ instead of *null* for E.

Similarly, if C is a required element in the target, it will be created by a different function $f_{\mathtt{C}}[\mathtt{A},\mathtt{B}]$. This function does not depend on the value of the *source* element C. Thus, even if in the source there may exist two tuples with the same combination for A and B, but with two different values for C (e.g. C is spouse, and an employee could be listed with two spouses), in the target there will be only one tuple for the given combination of A and B (with one, unknown, spouse). Thus, the semantics of the target is given solely by the values of the source elements that are *mapped*. Of course, a new correspondence from C to C will change the mapping: the employee with two spouses will appear twice in the target and E will be $f_{\mathtt{E}}[\mathtt{A},\mathtt{B},\mathtt{C}]$.

A similar mechanism for creation of new values in one target relation is adopted by the semantics of ILOG [HY90]. Next, we generalize this mechanism.

The second frequent case that requires creation of non-null values is that of a foreign key. In Figure 4.5(b), the target element C is stored in a different location (the set Spouses) than that of elements A and B. However, the association between A, B values and C values is meant to be preserved by a foreign key constraint (E plays the role of a pointer in this case). Our semantic translation recognizes such situations by computing the logical relation L(A, B, E, C) that joins Emps and Spouses, and should satisfy the inclusion

dependency $\texttt{Emps}_1[\texttt{ABC}] \subseteq \texttt{L}[\texttt{ABC}]$. However, this does not give a value for the required element E. As in the case of a single target set (but this time with a logical relation instead), we assign a function $f_{\texttt{E}}[\texttt{A}, \texttt{B}, \texttt{C}]$ to create values for E. The generated rule is:

foreach $\texttt{Emps}_1\ x$
let $a{=}x.\texttt{Emp.A}$, $b{=}x.\texttt{Emp.B}$, $c{=}x.\texttt{Emp.C}$
exists $\texttt{Emps} \leftarrow \langle\ \texttt{Emp} = \langle\ \texttt{A} = \text{a}, \texttt{B} = \text{b}, \texttt{E} = f_{\texttt{E}}[a, b, c]\ \rangle\ \rangle$,
$\texttt{Spouses} \leftarrow \langle\ \texttt{Spouse} = \langle\ \texttt{E} = f_{\texttt{E}}[a, b, c], \texttt{C} = c\ \rangle\ \rangle$

Briefly, the meaning of the rule is the following: for each (a, b, c) triple of values from the source, create an element Emp in Emps and create an element Spouse in Spouses, with the given source values (a, b, and c) in the corresponding places and with the *same* invented value $f_{\texttt{E}}[a, b, c]$ in the two places where the element E occurs. If duplicate (a, b, c) triples occur in the source (maybe with different D values) only one element is generated in each of Emps and, respectively, Spouses. Thus, we eliminate duplicates in the target.

In general, when the level of nesting is one (i.e., a flat schema), the rule used to associate Skolem functions with needed elements is as follows. (The fully nested case is presented in Section 4.4.3.)

Skolemization for atomic elements: *Given a mapping, each needed element in the target is computed by using a (different) one-to-one function that depends on all the mapped atomic elements of the target logical relation.*

The presence of functional dependencies in the target schema may change the functions used for value creation. To illustrate, if C is a key in Spouses (i.e., the functional dependency $\texttt{C} \rightarrow \texttt{E}$ holds), then in the above rule we replace $f_{\texttt{E}}[a, b, c]$ with $f_{\texttt{E}}[c]$.

4.4.2 Grouping of Nested Elements

Consider Figure 4.6(a), in which the target schema is nested on two levels: elements A and C are at the top level, while a Bs element can have multiple B sub-elements (Bs is of set type). Elements A, B, and C of the source $\texttt{Emps}_s$ are mapped, via correspondences,

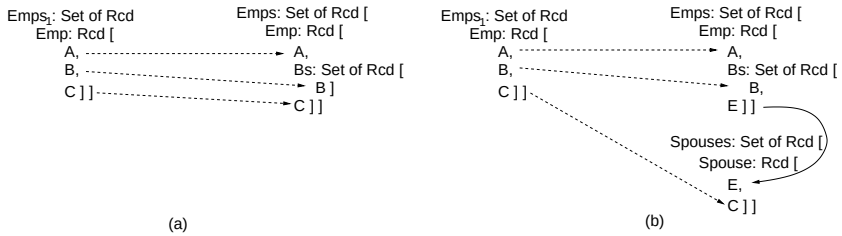


Figure 4.6: Grouping of elements in the target.

into the respective elements of the target `Emps`. The mapping requires that all (`A`, `B`, `C`) values found in the source be moved to the target. However, in addition to this, a natural interpretation of the target schema that we consider is that all the different b values are grouped together, for fixed values of `A` and `C`. For illustration, if `A`, `B`, and `C` are employee, child, and spouse names, respectively, the mapping requires the grouping of all children under the same set, if the employee and the spouse are the same. This behavior is not specified by the mapping (which is stated at the level of flat logical relations). Thus, one of the tasks of data translation is providing the desired grouping in the target. Grouping is performed at every nesting level, as dictated by the target schema. The property that the resulting target instance will satisfy is a well-known one: Partitioned Normal Form (PNF) [AB86].

PNF: *In any target nested relation, there cannot exist two distinct tuples that coincide on all the atomic elements (whether from source or created).*

To achieve this behavior, we use Skolemization as well. If a target element has set type, then its identifier (recall that each set is identified in the data model by a set id) is created via a Skolem function. This function *does not* depend on any of the atomic elements that are mapped under the respective set type, in the schema hierarchy. Instead it depends on all the atomic elements at the same level or above (up to the root of the schema). The same Skolem function (for a given set type of the target schema) is used across all mappings. Intuitively, we perform a deep union of all data in the target independently of their source. For the example of Figure 4.6(a), we create the rule:

foreach $\texttt{Emps}_1\ x$
let $a{=}x.\texttt{Emp.A}$, $b{=}x.\texttt{Emp.B}$, $c{=}x.\texttt{Emp.C}$
exists $\texttt{Emps} \leftarrow \langle\ \texttt{Emp} = \langle\ \texttt{A} = a,\ \texttt{Bs} = f_{\texttt{Bs}}[a, c],\ \texttt{C} = c\ \rangle\ \rangle,$
$f_{\texttt{Bs}}[a, c] \leftarrow \langle\ \texttt{B} = b\ \rangle$

The meaning of the above rule is the following: for each (a, b, c) triple of values from the source, create first (if not already there) a sub-element `Emp` in `Emps`, with the appropriate `A` and `C` sub-elements, and with a `Bs` sub-element, the value of which is the set id $f_{\texttt{Bs}}[a, c]$. Thus, the Skolem function $f_{\texttt{Bs}}$ is used here to create a set node. Also, we create (if not already there) a sub-element `B`, with value b, under $f_{\texttt{Bs}}[a, c]$. Later on, if a triple with the same `A` and `C` values (i.e., the values of a, c) but different `B` value (b') is retrieved from the source, then we skip the first creation step (the required `Emp` sub-element already exists). However, the second part of the **exists** clause applies, and we *append* a new `B` sub-element with value b' under the previously constructed set node $f_{\texttt{Bs}}[a, c]$. This mechanism achieves the desired grouping of `B` elements for fixed `A` and `C` values. A similar grouping mechanism can be expressed in XML-QL [DFF+98] using Skolem functions. It can also be implemented for languages that do not support Skolem functions like XQuery or XSLT.

4.4.3 Value Creation Interacts with Grouping

To create a nested target instance in PNF, we need to refine the process of creation of new values, which was described in Section 4.4.2 for the non-nested case only. We again explain our technique with an example. Consider Figure 4.6(b), where the elements `A` and `C` are stored in separate target sets. The association between `A` (e.g., employee name) and `C` (e.g., spouse name) is preserved via the foreign key `E` (spouse id). Thus, `E` is a required element and must be created. However, in this case, it is rather intuitive that the value of `E` should not depend on the value of `B`, but only on the `A` and `C` value. This, combined with the PNF requirement, means that all the `B` (child) values are grouped together if the employee and spouse names are the same. We achieve therefore the same

effect that the mapping of Figure 4.6(a) achieves. In contrast, if E is created differently based on the different B values, then each child will end up in its own singleton set. In our implementation of the logical mapping, we choose the first alternative, because we believe that this is the more natural interpretation. Thus, we adjust the Skolemization scheme of Section 4.4.1 as follows.

The function used for creation of an atomic element E *does not depend on any of the atomic elements that occur at a lower level than* E*, in the target schema.*

For the example of Figure 4.6(b) we create the rule:

foreach $\texttt{Emps}_1\ x$
let $a = x.\texttt{Emp.A}$, $b = x.\texttt{Emp.B}$, $c = x.\texttt{Emp.C}$
exists $\texttt{Emps} \leftarrow \langle\ \texttt{Emp} = \langle\ \texttt{A} = a,\ \texttt{Bs} = f_{\texttt{Bs}}[a, c, f_{\texttt{E}}[a, c]],\ \texttt{E} = f_{\texttt{E}}[a, c]\ \rangle\ \rangle$
$f_{\texttt{Bs}}[a, c, f_{\texttt{E}}[a, c]] \leftarrow \langle\ \texttt{B} = b\ \rangle,$
$\texttt{Spouses} \leftarrow \langle\ \texttt{Spouse} = \langle\ \texttt{E} = f_{\texttt{E}}[a, c],\ \texttt{C} = c\ \rangle\ \rangle$

As an extreme but instructive case, suppose that in Figure 4.6(b) we remove the correspondences that involve A and C, but keep the one that involves B. If A and C are needed, then they will be created by Skolem functions with no arguments. This is the same as saying that they will be assigned some constant values. Consequently, the two target sets Emps and Spouses will each contain a single element: some unknown employee, and some unknown spouse, respectively. In contrast, the nested set Bs will contain all the B values (all the children listed in $\texttt{Emps}_1$). Thus, the top-level part of the schema plays only a structural role: it is *minimally* created in order to satisfy the schema requirements, but the respective values are irrelevant. But this is fine, since there is nothing mapped into it. Later on, as correspondences may be added that involve A and C, the children will be separated into different sets, depending on the mapped values.

Example 4.33 *To illustrate the situation that was just described, consider the mapping example of Figure 4.7. The source schema describes companies that have an identifier, a name and a city. The target schema describes organizations (i.e., companies) but the organizations are nested in groups, with one group per city. Assume that the only value*

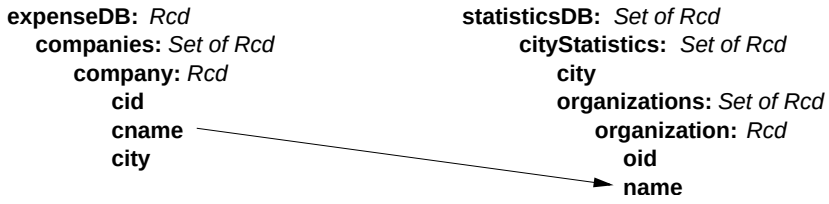


Figure 4.7: A Mapping example that requires grouping under a single set.

correspondence that has been specified by the user is the one that maps company names to organization names. Hence, for every company name, an organization will be generated in the target schema instance. However, each organization needs to be nested under a `cityStatistics` *element. A cityStatistics element is for a specific* `city`*. However, we do not know under what city an* `organization` *will be placed since the user has not mapped the* `city` *element from the source name to the target. This means that, for the specific mapping example, a single* `cityStatistics` *element will be created, with a* `city` *element specified by a Skolem function value, and all the organizations will be placed under that element. On the other hand, if the user adds one extra correspondence that maps* `city` *to* `city` *then the organizations will be placed under the right city.* ■

4.4.4 Skolem Function Generation

The generation of the Skolem functions sketched in the previous section can be done in time $O(N^2)$, where N is the size of the schema. This can be achieved by the graph-walking algorithm described in this section.

For a given mapping m, a graph is constructed, called the *query graph*, as follows. A node is constructed for each schema root of the source and target schemas. Then, for every variable binding in the from clauses of mapping m, a node is created representing the schema element to which the expression (consisting only of one variable) refers. The set containing every expression that uses a schema root or a variable followed by a set of record projections is constructed. A node is constructed in the graph for every such

expression. Each node represents the schema element to which the expression refers. Note that there may be more than one node representing the same schema element, if there are two variables in the mapping that refer to the same element. Then, an edge is created between two nodes if there is an edge between the nodes representing the respective schema elements in the graph representation of the schema. The final result is a forest. For every equality condition $e_1 = e_2$ used in the where or the **with** clause of the mapping, an "equality" edge is created between the nodes representing the elements to which expressions e_1 and e_2 refer. The term "equality" is just a name we give to these edges to distinguish them from the parent-child relationship edges.

Once the graph is created, its nodes are annotated. There are two kinds of annotations. The Skolem function annotations and the parameters annotations. For the Skolem function annotation, a unique Skolem function is assigned to each node of the graph. If two nodes represent expressions that refer to different schema elements, they are assigned different Skolem functions, but if they refer to the same schema element they are assigned the same Skolem function. The parameter annotation phase associates a set of expressions with each node of the graph. First, every node that corresponds to a source schema element is assigned the expression it represents. The next step is to propagate these annotations to the target schema nodes. (The steps below are also propagating these expressions to source schema nodes, but this information is ignored.) First, for every set of nodes that are connected through equality edges, their annotation becomes the same and equal to the annotation of one of the source nodes in that set, if there is one. Then for each target schema node that has been annotated during the previous phase, its value is propagated in one of the following three ways:

1. **Propagate up the record projection edges:** Every expression that a node acquires is propagated to its parent node if that parent node represents a record type schema element.

2. **Propagate down the record projection edges:** Every expression that a node acquires is propagated to its child nodes if they do not already have it, and if they are not related through a sequence of "equality" edges to a source schema node.

3. **Propagate equality edges:** Every expression that a node acquires is propagated to the nodes related to it through "equality" edges.

4. **Propagate down the $\in$-edges:** Every expression that a node corresponding to a set type schema element acquires is propagated to its child node if it does not already have it.

The propagation rules are applied repeatedly until no rules can be applied. The resulting annotated graph will help in transforming the mapping m to a query that will be executed over the source data and transform it to data conforming to the target schema.

4.4.5 Transformation Queries Generation

Once the query graph of a mapping has been generated, it can be converted to an actual query in the target query language. This means that, if the source data is relational data, the query will be a relational SQL query. If it is XML data, then the query will be XQuery or XSLT. We describe here how a mapping can be converted to an XQuery. The XSLT and SQL approaches are similar. The difference for SQL is that it will never have nested elements. This section describes how this transformation can be achieved and corresponds to the box labeled "Query Generation" in Figure 4.1.

Let Q be the **foreach** clause of a mapping m. It is straight-forward to translate that clause to an XQuery with a **for** and where (but no **return**) clause. Let Q_f be that query. The generation starts by a target schema root in the query graph and performs a depth first traversal.

- If the node being visited corresponds to a complex type schema element, the XML tags with that element name are generated and its children are visited. The results produced by visiting its children will be between the starting and closing tag of that node.

- If a node corresponding to an atomic type element is visited, then: (i) if the node is related through an "equality" edge path to a source schema node, then the opening and closing tags are created and the value between them is the expression represented by the source schema node; (ii) if the node corresponds to an optional element, nothing is generated; (iii) if the node corresponds to a nullable schema element, the null value is generated; and finally, if none of the above applies, a value is generated for that element. The value is generated by calling the unique Skolem function assigned to that node of the query graph with arguments that include the expressions that are in the annotation of that node.

- If the node is a set node, a **for**-where return query is produced, which is the query Q_f with its variables subscripted with the nesting depth of the query. In addition, an extra where clause binary equality condition is added in the where clause that associates the expression in its annotation with the corresponding expressions of the surrounding query, creating a correlated query. A correlated query is a query that is used within another query and is related to it through a condition in its where clause.

The SQL and XSLT versions of the queries are generated in a similar way. It is only a syntactic difference. However, an issue in this process is how the resulting data from different mappings can be merged. This can be achieved in XSLT by exploiting the XSLT feature of producing keys on the fly for transformed elements. In particular, two XSLT scripts can be used. The first transforms the data elements into a "flat" view and also annotates them with Skolem functions that identify the set to which they belong. The

second XSLT script can then take this result and merge the data into the right sets, by using the set identifier information produced by the first XSLT script.

Example 4.34 *Figure 4.8 shows the annotated query graph for the following mapping*

m: **foreach** S.grants g, g.grant.sponsor→government r, S.projects p, S.contacts c
where p.project.name=g.grant.project and c.contact.cid=r
exists S.govProjects j, j.govProject.fundings u, T.finances f, T.companies o
where j.funding.finId=f.finances.finId and f.finances.company=o.company.cname
with j.govProject.code = p.project.name and u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and f.finances.amount=g.grant.budget and
f.finances.sponsorPhone=c.contact.phone

between the schemas S *and* T *of Figure 3.1. In the Figure, all the nodes that correspond to source schema elements have been omitted, apart from those that are connected through "equality" edges with nodes corresponding to elements of the target schema. These nodes are indicated with square shapes, while all the nodes corresponding to source schema elements are ellipses. The annotation of each node is determined by the annotation of its incoming edge from the parent node. Each annotation is one or more lines. The first line is the unique Skolem function that has been assigned to the node and the remaining lines contain the expressions that result from of the propagation phase.*

The XQuery indicated in Figure 4.9 is the XQuery generated from that graph. The algorithm starts at the target schema root node, and generates by default the elements <T> and </T> (lines 3 and 54). Then the child nodes of the root node are visited, and whatever is to be generated is generated between these nodes. For example, when node govProjects *is visited, the algorithm finds that it is a* Set *type, so it produces a source query that is the query Q_f as described previously (lines 4-09). Then its child node* $x4$.govProject *is visited and the element <govProject> is created (lines 10 and 27). The specific element is a* Rcd *type, so child elements are produced for each child of that node. In particular, when the node* $x4$.govProject.code *is visited, since it is an atomic type, line 11 is output. The element <code> comes from the attribute label,*

while the value comes from the annotation $x0$.`grant.project` *of the node (modified in an XQuery expression syntax). Then the* $x4$.`govProject.year` *node is visited. The algorithm detects that it has a Skolem function annotation, and checks whether it can be null or not. Since it cannot, it outputs an XML element having as a value the Skolem function with arguments the expressions that are in the annotation, hence producing line 12. When the node* $x4$.`govProject.fundings` *is visited, the algorithm finds that it is a set type, and instead of generating an element, it produces a subquery that will generate the values of that set. The query is query* Q_f *but with its variable names extended with the string "L0", since this is the first level of nesting (lines 13-19 apart from 18 are produced). To keep the correlation with the parent query, the condition of line 18 is generated. The expressions that should be equal between the two queries are determined by the annotation of the node* $x4$.`govProject.fundings` *(without the Skolem function name), which represents the set. The same process continues until the depth first traversal of the tree is finished. The final generated query is in Figure 4.9.* ■

4.5 Analysis

4.5.1 Complexity and Termination of the Chase

The chase is the main process used to construct the logical associations. In the general case, checking whether an association can be chased with a dependency is exponential in the size of the dependency [PT99] and the size of the association. If the dependency is an NRI, then the complexity of checking whether an association can be chased with the NRI is linear in the size of the NRI and the size of the association. The size of an NRI or an association is determined by the number of variables and the number of equalities it contains. The reason is that NRIs are a special case of nested dependencies in which matching involves paths on the schema. The size of an association is linear in the size

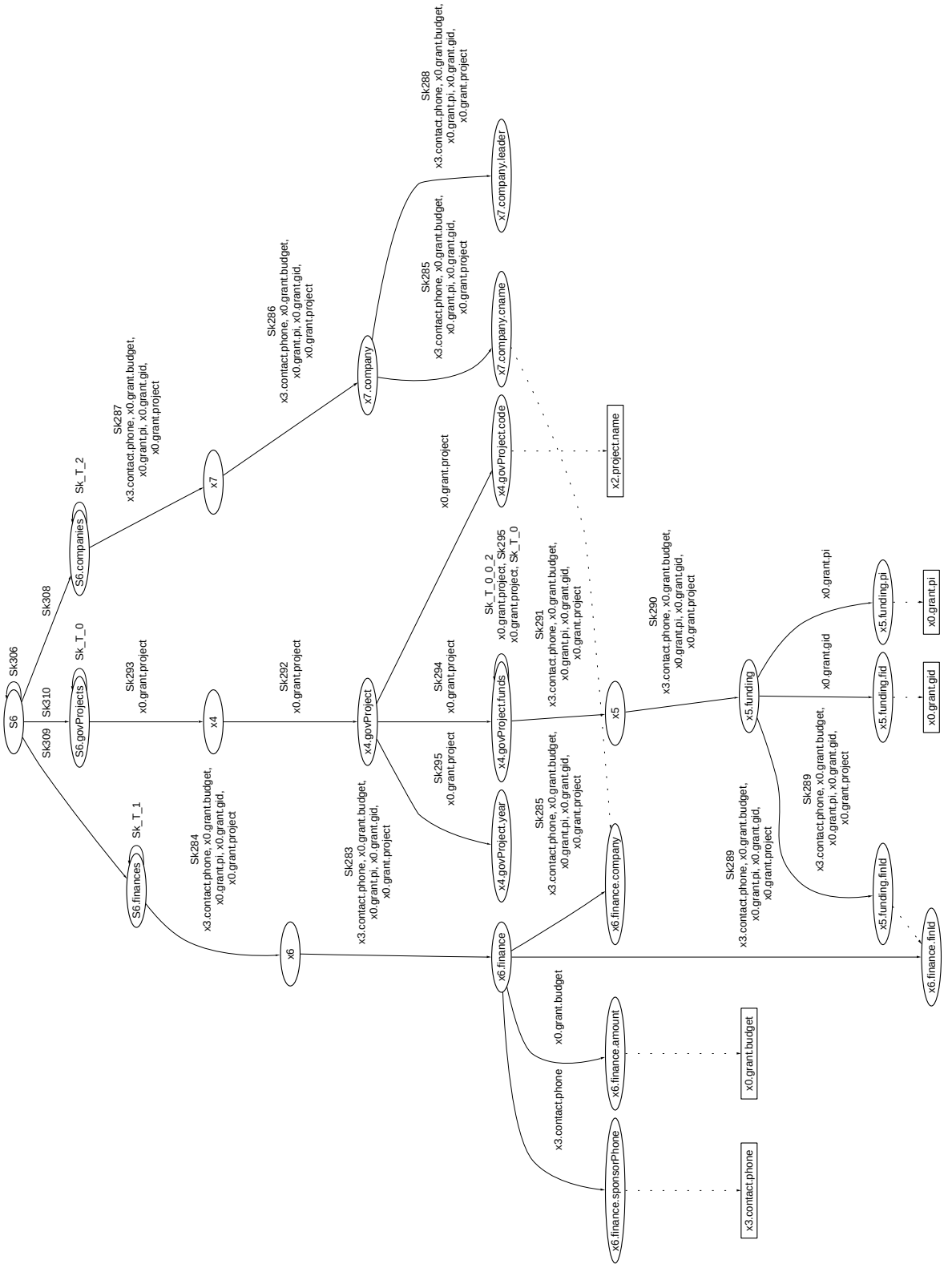


Figure 4.8: An annotated query graph.

```
(01) LET $doc0 := document("input XML file goes here")
(02) RETURN
(03) <T>
(04)   {distinct-values (
(05)   FOR $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/government,
(06)       $x2 IN $doc0/S/project, $x3 IN $doc0/S/contact
(07)   WHERE
(08)      $x2/name/text() = $x0/project/text() AND $x1/text() = $x3/cid/text()
(09)   RETURN
(10)      <govProject>
(11)         <code>  {  $x0/project/text() }  </code>
(12)         <year>  {  "Sk295(", $x0/project/text(), ")" }  </year>
(13)         {distinct-values (
(14)         FOR $x0L1 IN $doc0/S/grant, $x1L1 IN $x0L1/sponsor/government,
(15)             $x2L1 IN $doc0/S/project, $x3L1 IN $doc0/S/contact
(16)         WHERE
(17)            $x2L1/name/text()=$x0L1/project/text() AND $x1L1/text()=$x3L1/cid/text()
(18)            AND $x0/project/text() = $x0L1/project/text()
(19)         RETURN
(20)            <funding>
(21)               <fid> {$x0L1/gid/text()} </fid>
(22)               <pi> {$x0L1/pi/text()} </pi>
(23)               <finId>{"Sk289(", $x3L1/phone/text(), ", ", $x0L1/budget/text(), ", ",
(24)                        $x0L1/pi/text(), ", ", $x0L1/gid/text(), ", ",
(25)                        $x0L1/project/text(),")"}</finId>
(26)            </funding> ) }
(27)      </govProject> ) }
(28)   {distinct-values (
(29)   FOR $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/government,
(30)       $x2 IN $doc0/S/project, $x3 IN $doc0/S/contact
(31)   WHERE
(32)      $x2/name/text() = $x0/project/text() AND $x1/text() = $x3/cid/text()
(33)   RETURN
(34)      <finance>
(35)         <finId>{"Sk289(",$x3/phone/text(),",",$x0/budget/text(),",",$x0/pi/text(),
(36)                        ",",$x0/gid/text(),",",$x0/project/text(),")" }  </finId>
(37)         <amount> {$x0/budget/text()} </amount>
(38)         <sponsorPhone> {$x3/phone/text()} </sponsorPhone>
(39)         <company>{"Sk285(",$x3/phone/text(),",",$x0/budget/text(),",",$x0/pi/text(),
(40)                          ",",$x0/gid/text(),",",$x0/project/text(),")"} </company>
(41)      </finance>)}
(42)   {distinct-values (
(43)   FOR $x0 IN $doc0/S/grant, $x1 IN $x0/sponsor/government,
(44)       $x2 IN $doc0/S/project, $x3 IN $doc0/S/contact
(45)   WHERE
(46)      $x2/name/text() = $x0/project/text() AND $x1/text() = $x3/cid/text()
(47)   RETURN
(48)      <company>
(49)         <cname>{"Sk285(", $x3/phone/text(), ", ", $x0/budget/text(), ", ", $x0/pi/text(),
(50)                        ", ", $x0/gid/text(), ", ", $x0/project/text(),")"} </cname>
(51)         <leader>{"Sk288(", $x3/phone/text(), ", ", $x0/budget/text(), ", ", $x0/pi/text(),
(52)                        ", ", $x0/gid/text(), ", ", $x0/project/text(),")"} </leader>
(53)      </company>)}
(54) </T>
```

Figure 4.9: A generated XQuery.

of the schema. Hence, a chase step takes time linear in the size of the schema and the constraints.

Unfortunately, it is known that, in the general case, the chase with tuple generating dependencies may not terminate. NRIs generalize the class of relational inclusion dependencies, and, similarly, our chase is a generalization of the relational chase. NRIs are also a special case of the embedded path-conjunctive dependencies (EPCDs) used by Popa and Tannen [PT99] to express constraints in nested and object-oriented schemas. The chase that we use to compute logical relations is a special case of the chase with EPCDs. Even for this special case, the chase may not terminate [Pop00]. One possible solution to this problem is to restrict the class of legal constraints so that the chase is guaranteed to terminate.

Next, we generalized the notion of acyclicity for the case of nested schemas and constraints. The class of acyclic NRIs guarantees that any chase sequence of an association terminates and, moreover, it does so in quadratic time in the number of the NRIs. Termination of chase ensures that there are only finitely many logical paths (and logical associations) for a schema, and therefore the search space of all possible interpretations of the correspondences is finite.

However, inspired by weakly recursive ILOG [HY90], we allow a special form of cyclicity, hence, considering instead of acyclic NRIs, a special class called *weakly recursive* NRIs. This class is strictly more expressive (when the data model is relational) than acyclic inclusion dependencies, but the limited form of cyclicity that is allowed is enough to guarantee that the chase always terminates. While there are bad cases of real-schema constraints that are cyclic and not weakly recursive, there are also useful constraints that fit into the class of weakly recursive NRIs even though they are cyclic, for example, inverse relationship constraints [BDK92], which are quite common in object-oriented schemas and potentially useful in XML schemas. Intuitively, if a relationship R associates an object (or concept) x with a object (or concept) y, the inverse relationship associates

the object (or concept) y with the object (or concept) x.

To define weakly recursive NRIs, consider a set Σ of NRIs over a nested relational schema and el_1, el_2, ..., the atomic type elements occurring in the schema. Let **foreach** P_1 **exists** P_2 **with** C be an NRI constraint. We say that element el_j is *determined* by element el_i with respect to NRI d and write $el_i \xrightarrow{d} el_j$, if C contains the equality $e_2{=}e_1$ where e_1 starts with a variable from P_2 and refers to element el_j while e_1 uses a variable of P_1 and refers to el_i. Every element refered to by a valid expression e_u that starts with a variable of P_2, but is not determined by any element, is said to be *undetermined* with respect to the NRI d. Following weakly recursive ILOG [HY90], we have the following definition:

Definition 4.35 (Weakly Recursive Constraints) *Given a set Σ of NRI constraints and a schema $\mathcal{S}$, compute a directed labeled graph G_Σ as follows:*

1. *For each atomic element in $\mathcal{S}$ create a graph node labeled "white".*
2. *For each NRI d in Σ*
   - *If $el_i \xrightarrow{d} el_j$ add an edge from el_i to el_j.*
   - *For each undetermined element el_k with respect to d, add an edge from el_j to el_k and label el_k "black".*

The schema $\mathcal{S}$ with the set Σ or NRIs is weakly recursive if every cycle that exists in G_Σ consists of white nodes only.

Example 4.36 *The following example illustrates cyclic NRIs that are not weakly recursive. The example uses the relational data model for notational simplicitly. Consider a relational schema with two tables* `Dept(dno,dname,mgr)` *and* `Emp(eno,ename,dep)`*, and assume two foreign key constraints: r_1 from* `dep` *of* `Emp` *referring to* `dno` *of* `dept`*, and r_2 from* `mgr` *of* `Dept` *referring to* `eno` *of* `Emp`*. Then r_1 and r_2 can be expressed as a set of two NRIs that are not weakly recursive. The chase with them never terminates. The fact that*

the chase never terminates means that there may be cases with infinitely many interpretations for a given set of correspondences. If the above schema with the departments and the employees is the source schema, consider a target schema consisting of one relation with only two attributes: `Manages(emp, mgr)`*. Assume two correspondences, one from element* `eno` *of* `Emp` *to* `emp` *of* `Manages` *and one from* `eno` *of* `Emp` *again to* `mgr` *of* `Manages`*. The first interpretation one may think of is that the employee number of an employee and the employee number of her manager are recorded in the* `Manages` *relation. Another interpretation is that the employee number of an employee and the employee number of the manager of her manager are recorded. A third interpretation is that the employee number of an employee and the employee number of the manager of the manager of her manager. This inference can go on forever. The graph showing that this is not a weakly recursive case can be found on the left of Figure 4.10 where the gray nodes represent the black colored nodes.*

■

Example 4.37 *This example illustrates constraints that look cyclic, but in fact are weakly recursive according to our definition. Consider the following schema consisting of two nested tables:*

```
Company: Set of Rcd [
       cid : String,
       cname : String,
       projects: Set of Rcd [
              project : String,
              amount : String ] ]

Project: Set of Rcd [
       name : String,
       year : String,
       companies: Set of Rcd [
              cid : String ] ]
```

The following two NRIs express the fact that there exists a many-to-many inverse relationship between companies and projects:

i_1: **foreach** company c. c.projects p
exists project p', p'.companies c'
with p'.name=p.project and c'.cid=c.cid

i_2: **foreach** project p', p'.companies c'
exists company c, c.projects p
with p'.name=p.project and c'.cid=c.cid

One can verify that the set $\{i_1, i_2\}$ is a set of weakly recursive NRIs. Thus the chase can be safely used for this schema to compute all the logical associations needed by the mapping algorithm. The graph showing this can be found on the right of Figure 4.10. Indeed, the structural associations of these two schemas are:

A^1: select * from company c
A^2: select * from company c, c.projects p
A^3: select * from project p'
A^4: select * from project p', p'.companies c'

When chased with constraints i_1 and i_2, associations A^1 and A^3 remain unaffected since neither i_1 nor i_2 can be applied. Associations A^2 can be chased only once with i_1 and A^4 can also be chased only once with i_2. The resulting associations are:

A^1_l: select * from company c
A^2_l: select * from company c, c.projects, p, project p', p'.companies c'
where p'.name=p.project and c'.cid=c.cid
A^3_l: select * from project p'
A^4_l: select * from project p', p'.companies c', company c, c.projects
where p'.name=p.project and c'.cid=c.cid

■

Fagin et al. [FKMP03a] have shown the following theorem.

Theorem 4.38 *Chase with a finite set of weakly recursive NRIs always terminates, and it does so in time quadratic in the size of NRIs.*

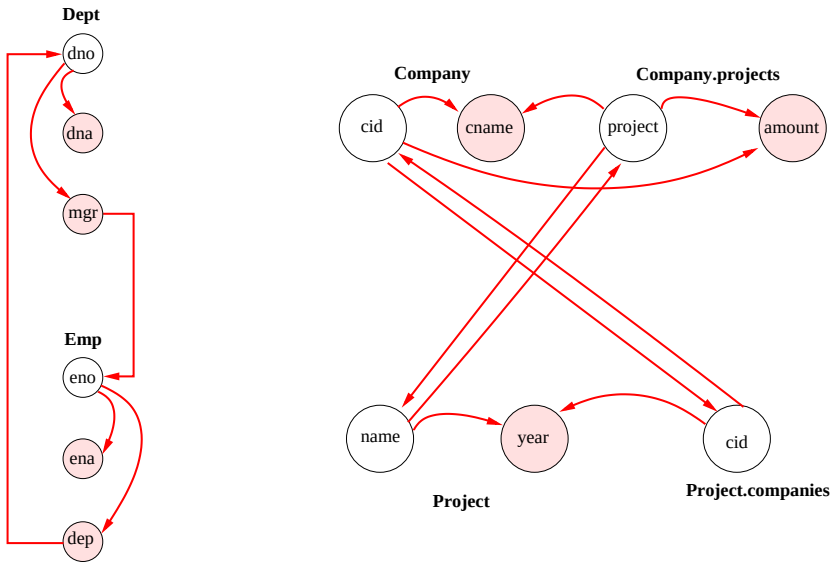


Figure 4.10: Graphs showing non-weakly (left) and weakly recursive (right) NRIs

Another form of cyclicity that is often found in schemas is type recursion. Type recursion occurs when a complex type is used in its own definition. Type recursion leads to infinitely many structural associations that may result in infinitely many possible interpretations of a given set of correspondences. Figure 4.11 illustrates how type recursion can lead to an infinite number of mappings. In order to partially cope with this limitation in our implementation (presented at the end of this chapter), we allow recursive types but we require the user to specify the recursion level, or we have it bounded by a specific constant value. For example, in the mapping scenario of Figure 4.11, if we bind the recursion level to one, then only the first two mappings shown in the figure will be generated by the system, and the user will be left to select the one that better describes the intended semantics of the correspondences.

4.5.2 Completeness and Correctness of Semantic Translation

Theorem 4.39 *Algorithm 2 computes all the semantically complete mappings.*

Proof. If there is a semantically complete mapping m with the source and target logical associations $A^{\mathcal{S}}$ and $A^{\mathcal{T}}$, since the algorithm considers all the pairs of source and target

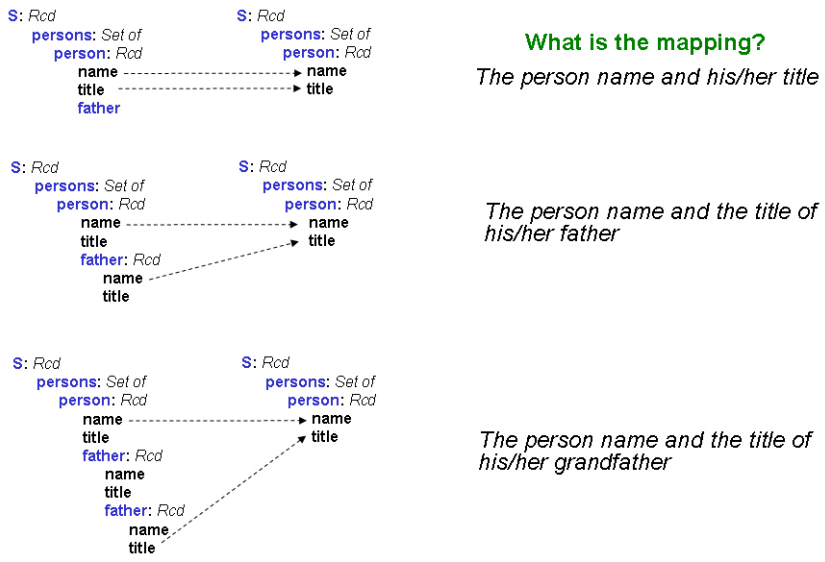


Figure 4.11: Unlimited candidate mappings due to recurvive types.

associations, it will consider $<A^{\mathcal{S}}, A^{\mathcal{T}}>$ too. Furthermore, the algorithm also considers all the ways that the correspondences are covered, hence, it will consider the way of coverage through which the covered correspondences of m are covered. Consequently, mapping m would have been produced by Algorithm 2. ∎

It is clear that the chase, by computing logical associations, finds relationships between schema elements. What is not so clear is whether *all* the natural relationships are discovered by the algorithm. In order to study this issue and characterize these relationships, we focus on the case where the schema is relational. This focus allows us to use some results from the research on the universal relation [MUV84]. Since schemas will be relational, we will use the term "attributes" to refer to schema elements in a relation. Furthermore, for the current subsection, when we refer to a logical association A of a schema S, we mean its instance relation. Recall that an association has been defined as a query. The instance relation of association A, with respect to an instance $\mathcal{I}$ of schema S, is the result of evaluating the association query over the instance $\mathcal{I}$.

Consider a schema consisting of several relations. All the attributes in the schema are assumed to have unique names, with one exception: attributes involved in foreign

key constraints must have the same name. Thus, all the information in the database is in the attributes: each reflects a unique feature of the data, and hence has a unique name. Let U be the universe of all such unique attributes in the schema, and let $\mathcal{I}$ be an instance over the set of database relations. We assume that $\mathcal{I}$ satisfies all the foreign key constraints (NRIs) *and* the corresponding key constraints. Moreover, let Δ be the set of all functional dependencies that are obtained by "lifting" each key constraint from a functional dependency on the universe of its respective relation to a functional dependency on the universe U.

Following the terminology of Maier et al. [MUV84], a *weak universal relation (or weak UR)* associated with $\mathcal{I}$ and Δ is defined as any relation u over U with the following properties: For every relation R of the original schema, 1) $\Pi_R(u)$ is a superset of the current relation for R in $\mathcal{I}$, and 2) u satisfies Δ. Thus the weak universal relation can be viewed as some "idealized" completion of the database relations in $\mathcal{I}$, satisfying Δ. It provides a view of the underlying database that is independent of any particular database design scheme. However, there are (infinitely) many possible weak URs, and we cannot know which one truly represents the real world. Therefore, the only facts that can be deduced about the weak UR from the given database relations are those facts that are true for *all* weak URs. Maier et al. [MUV84] defined the *connection* on a subset of attributes X of U, with respect to Δ, denoted by $[X]^{\Delta}$, as the set of tuples t such that for *every* weak UR u, there is tuple t' in u that agrees with t on X. Maier et al. also showed that $[X]^{\Delta}$, for a given $\mathcal{I}$, can be computed by materializing first a *representative* weak UR, called RI_{Δ}, as follows: each tuple of each relation in $\mathcal{I}$ is inserted in RI_{Δ}, after it has been padded with new "null" symbols. Then the resulting relation is chased to enforce the satisfaction of the dependencies in Δ. The result of the chase is RI_{Δ}. The connection $[X]^{\Delta}$ is then equal to the projection of RI_{Δ} on X followed by the elimination of any tuples that contain "null" symbols. In practice, we want $[X]^{\Delta}$ to be computed by efficient queries without having to chase instances. The general decision problem of

whether there exist such queries is not known to be solvable. However, if we consider in Δ only the dependencies that arise from the key and foreign key constraints, then $[X]^{\Delta}$ is computable from the logical associations.

Theorem 4.40 *For every instance $\mathcal{I}$ of a schema S, and for every set of attributes $X \subseteq U$, if $A_1, \ldots, A_n$ are the logical associations of schema S over the instance $\mathcal{I}$, then,* $[X]^{\Delta} = \Pi_X(A_1) \cup \ldots \cup \Pi_X(A_n)$.

Proof. ($\Rightarrow$) Let $t \in [X]^{\Delta}$ and t has no null attribute. If all the attributes of X are in the same relation R, then for every logical association A_k that was generated by chasing R we will have $t \in \Pi_x(A_k)$. If X are attributes that are related through a join path, then due to the way associations are constructed (through the chase), there will be at least one logical relation A_k having all attributes X for which $t \in \Pi_x(A_k)$. If t has some null attributes, let $X' \subset X$ such that $\Pi_{X'}t$ has no null attributes. Then, following the previous reasoning, there will be at least one logical association for which $t \in \Pi_{X'}(A_k)$. Association A_k can be padded with nulls for the attributes of X that are not in A_k. Then, we will have that $t \in \Pi_X(A_k)$. Hence, $[X]^{\Delta} \subseteq \Pi_X(A_1) \cup \ldots \cup \Pi_X(A_n)$. ($\Leftarrow$) If $t \in \Pi_X(A_1) \cup \ldots \cup \Pi_X(A_n)$ there is at least one logical association A_k for which $t \in \Pi_X(A_k)$. Let the attributes X'' be the not null attributes of t. Since attributes X'' are in the same logical association there is a set of attributes $X' \subseteq X''$ that consists of the attributes in the structural association from which association A_k was constructed through chase. By the definition of the logical association, the values of the other attributes $X'' - X'$ will be generated through the chase. But this is how the representative universal relation RI is constructed. Hence, $[X]^{\Delta} \supseteq \Pi_X(A_1) \cup \ldots \cup \Pi_X(A_n)$. ∎

Theorem 4.40 shows that each connection is fully computable by a union of projections of logical associations. These logical associations are in turn computed by the chase.

The importance of correspondences, as a method for specifying relationships between attributes of different schemas, has already been recognized in the research community [RB01, DR02, KN03]. In the mapping generation Algorithm 2, we precompute all

the logical associations, independently of any correspondences. Then, the correspondences are grouped together to form sets of associated attributes. Maximal such groups are discovered by checking coverage by some logical association. (Recall that logical association A *covers* a correspondence c, if the attributes of A include the attributes X used by the correspondence.)

Proposition 4.41 *Given two schemas $S^{\mathcal{S}}$ and $S^{\mathcal{T}}$, and a set of correspondences between their attributes, let $m_0, m_1, ..., m_k$ be the mappings generated by Algorithm 2. This set of mappings computes* all *the connections. In particular, let V be a subset of the correspondences, and $X^{\mathcal{S}}$ and $X^{\mathcal{T}}$ all the source and target attributes used in V. Then, for any given instance $I^{\mathcal{S}}$ and $I^{\mathcal{T}}$ of schema $S^{\mathcal{S}}$ and $S^{\mathcal{T}}$, respectively, for which every mapping m_i is satisfied, $[X^{\mathcal{S}}]^{\Delta} \subseteq [X^{\mathcal{T}}]^{\Delta}$.*

Proof. Mappings are generated through the coverage of a number of correspondences by the pair of source and target logical associations $<A_s, A_t>$. Each mapping is in the form $Q^{\mathcal{S}} \subseteq Q^{\mathcal{T}}$ where query $Q^{\mathcal{S}}$ (respectively, $Q^{\mathcal{T}}$) is the logical association A_s (respectively, A_t) followed by a projection on the attributes used by the covered correspondences. Each association, is actually the connection $[X^A]$ where X^A is the set of attributes used in the association. Thus, its projection on the set of attributes $X \subseteq X^A$ is also a connection.

We showed that the queries in the mappings we generate are computing connections of the attributes used in the correspondences. What is not clear, is whether there are other mappings that can also compute a connection. Assume that we were grouping together correspondences that are covered by different logical associations. Let A_{s1} and A_{s2} be two such logical associations in the source schema, A_t is a logical association in the target schema, and C_1, C_2 are the sets of correspondences covered by the pairs $<A_{s1},A_t>$ and $<A_{s2},A_t>$, respectively. Since A_{s1} and A_{s2} are different logical associations, there is no join path between them. Hence, if C_1 and C_2 had been grouped together to form a mapping, the source schema query of the mapping would have involved a cartesian

product of associations A_{s1} and A_{s2}. The results of that query would be $\Pi_{C_1C_2}(A_{s1} \times A_{s2})$. Such tuples cannot be generated as a union of projections on the logical associations as required by Theorem 4.40, hence the theorem would have been violated.

Similarly, consider correspondences c_1 and c_2 that are both covered by the pair of associations $<A_s, A_t>$. If correspondences c_1 and c_2 are considered separately by forming different mappings, then, it is not guaranteed that $[XZ]=\Pi_{XZ}A_s$, where X and Z the attributes of A_s used by c_1 and c_2, respectively.

Hence, from the 2^N ways that correspondences can be grouped together and form mappings, only groups formed through coverage by a logical association guarantee that connections are preserved. This is exactly what our mapping generation algorithm produces. In that sense, the algorithm is complete. ■

Of course, the mappings generated by Algorithm 2 are not the only meaningful mappings that are consistent with the correspondences and the schema constraints. Many other mappings may exist, but according to Theorem 4.40, it is not guaranteed that they will preserve the connections. In our approach, we chose to follow a mapping generation approach that generates mappings that preserve the connections. The reason is that connections are believed to be the most natural way for data administrators to provide meaningful units [Cod79].

An important observation about the semantic translation algorithm is that it abstracts away the specific schema details such as normalization and nesting. Theorem 4.40 is a precise statement of this. We could change the logical design (in the source and/or target schema), but we would get the same mapping, as long as the attributes have the same semantics associated with them, and the correspondences remain the same. Figure 4.12 explains graphically the computation that the mapping generation algorithm performs. The source and target logical associations generate flat relational views of the source and target schema, respectively, and the correspondences are used between these two relational views.

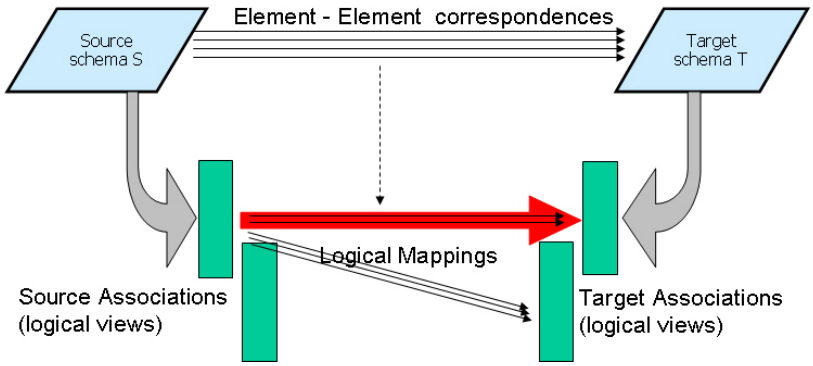


Figure 4.12: Graphic explanation of the computation of the mapping generation algorithm.

If every attribute of the schemas is involved in some correspondence, then each mapping generated by our mapping translation algorithm is different from all the other mappings. However, correspondences may be defined only for some schema attributes. For that case, the mapping generation algorithm may produce some mappings that are are logically implied by others. The following proposition provides one condition for when this can happen. The optimization of Section 4.3, illustrated in Figure 4.4, is based on Proposition 4.42.

Proposition 4.42 *A mapping m, defined by the pair of source and target logical associations $<A_s,A_t>$, is redundant if there is another mapping m', defined by the pair $<B_s,B_t>$ of source and target logical associations, that covers the same correspondences, and $B_s \dot{\preceq} A_s$ and/or $B_t \dot{\preceq} A_t$. A mapping in a set of mappings is redundant if it is logically implied by the other mappings in the set.*

Proof. Let two mappings that have the same target schema logical association but one uses B_s as the source schema logical association and the second uses A_s. if $B_s \dot{\preceq} A_s$, by definition, every tuple in the results of the logical association A_s over an instance $\mathcal{I}$ (recall that a logical association is a query), when projected on the attributes of B_s will give a tuple that exists in the results of the logical association B_s over the same instance $\mathcal{I}$. If X^{B_s} be the attributes of B_s, then $\Pi_{X^{B_s}} A_s(\mathcal{I}) \subseteq B_s(\mathcal{I})$, where $A(\mathcal{I})$

means evaluation of association A over the instance $\mathcal{I}$. Since association A_s does not cover any extra correspondence apart from what covered by B_s, $X^c \subseteq X^{B_s}$, where X^c is the set of attributes of B_s that participate in the covered correspondences. Hence, $\Pi_{X^c} A_s(\mathcal{I}) \subseteq \Pi_{X^c} B_s(\mathcal{I})$. Which means that the set of source instance values restricted by the mapping that uses A_s is a subset of the set of source instance values restricted by the mapping that uses B_s. Similar results apply for the target schema associations as well.

■

The notion of logical associations is similar to the notion of *maximal objects* [MU83]. The construction of maximal objects also considers multi-value dependencies. Logical associations, on the other hand, may generate larger groups of schema elements than maximal objects, and do not make the "one flavor assumption" of the universal relation. The one flavor assumption states that an attribute of the schema cannot have more than one meaning. However, the main principle behind maximal objects and logical associations is the same.

4.5.3 Characterization of Data Translation

Consider a data exchange setting, that is, a source schema $\mathcal{S}$, a target schema $\mathcal{T}$, a set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$, and an instance $\mathcal{I}$ of schema $\mathcal{S}$. The problem in data exchange is to find a *solution*, i.e., an instance of the target schema $\mathcal{T}$ that is consistent with the mappings in $\mathcal{M}$. As before, we assume that we are in the relational model, which allows us to use some of the existing results in the research literature. In a data exchange setting, there may be more than one solution. Fagin et al. [FKMP03b] have provide an algebraic specification that selects among all solutions of a data exchange setting, a special class that is referred to as a *universal solution*. A universal solution has been shown to have no more and no less data than required for data exchange, and represents the entire space of possible solutions. A universal solution I_u is a solution for which there is a homomorphism h to every other solution J. Fagin et al. provided an

algorithm, based on chase, for computing a universal solution. That algorithm starts with the source schema instance $\mathcal{I}$ and chases it with the mappings (recall, that mappings can be viewed as source-to-target constraints), and also with the target schema constraints. The results of the chase give a target schema instance that is a universal solution.

The data translation algorithm of Section 4.4 follows the algorithm described by Fagin et al. It considers each mapping as a constraint and applies it to the existing instance. The application of the mapping generates new data tuples in the target. These tuples consist of constant values and "fresh" values. The constant values are the values that have came from the tuples of the source schema and were specified in the mappings (i.e., constant values are the values in the target schema attributes that are used in correspondences). The "fresh" values are in the attributes whose values are not determined by the mappings. Our data translation algorithm generates for each different tuple of determined attributes a different "fresh" value for every undetermined attribute. This task is actually the chase using the mappings. The target instance generated through this process need not be chased also with the target schema constraints as Fagin et al. described. The reason is that the chase with the target schema constraints has already taken place during the creation of the logical associations that are embedded in the mappings. Consequently, the instance $\mathcal{I}$ generated by our data translation algorithm is a *universal solution*. Indeed, any other solution J will agree with the instance $\mathcal{I}$ on the elements for which correspondences have been defined (the elements determined by the mapping). For all the other elements, the data translation algorithm has generated Skolem values that can be mapped to the corresponding values of instance J. Thus, there is a homomorphism from $\mathcal{I}$ to any other solution J, which means that $\mathcal{I}$ is a universal solution.

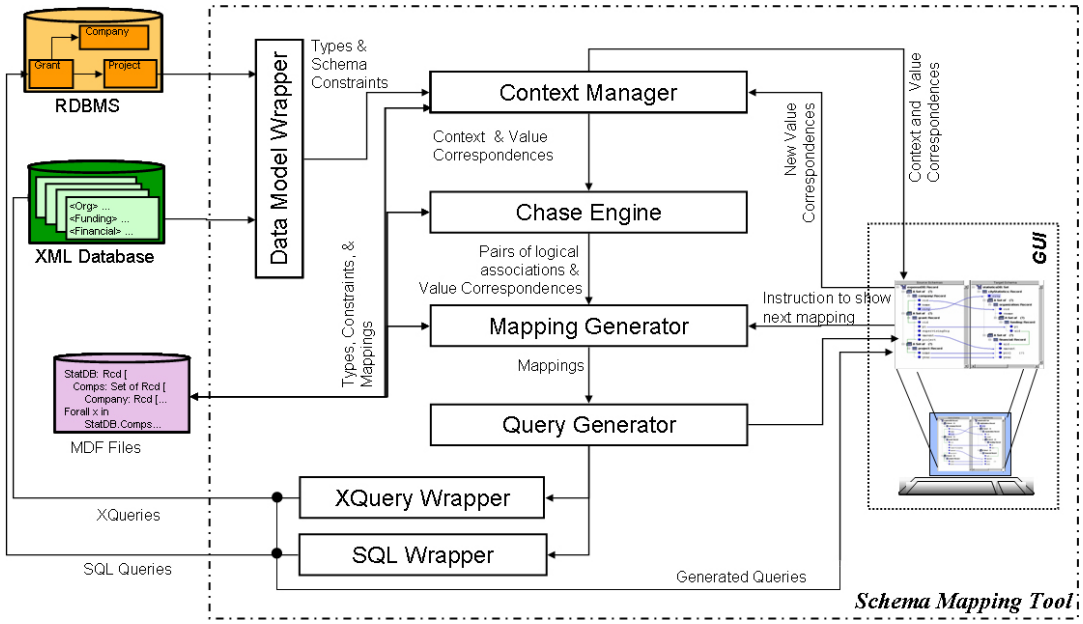


Figure 4.13: The Clio system architecture.

4.6 Implementation and Experimentation

4.6.1 Tool Description

We have implemented the solution described in the previous sections in a prototype system called *Clio*. The Clio tool is a mapping management tool that has been developed in collaboration with IBM Almaden Research Center. Clio has many features. One of them is data sampling. Data sampling is used to assist the user who defines the mappings in selecting those that best describe her intentions [YMHF01]. Clio also assists the user in finding correspondences between schema elements [NHT+02]. For each element, it analyzes the data values and derives a set of features. The overall feature set forms the characteristic signature of an attribute. Signatures can then be used to detect semantically similar elements across different schemas. Our main contribution to Clio is the development of the mapping generation engine. In the following paragraphs, we will describe the parts of Clio that we have developed. We will also give a brief description of the Clio GUI, since it is closely related to our work. Our contribution to the GUI

was the design and implementation of a component that allows the user to annotate correspondences with value translation functions.

The general architecture of Clio is shown in Figure 4.13. It is composed of various modules, each with a clearly specified task. In principle, modules could be replaced or upgraded without affecting the overall operation of the system. One of our major requirements was to make the whole mapping generation process incremental. Often, users will provide value correspondences incrementally and want to see partial results before adding additional correspondences. Thus, modules should avoid redundant recomputation and redundant exchange of data. Furthermore, since schema mapping is a large task, users may want to save the state of the system and continue at a later time from the point at which they stopped. The following is a description of each of the modules.

- **User Interface:** A user interacts with the system through a graphical user interface (GUI). Through the GUI, the user has the ability to view the schemas in a uniform representation, the value correspondences, and the generated mappings. At the beginning of the interaction with the system, the user loads the source and target schemas, which may be relational or XML Schemas. The schemas are wrapped by the system in the nested relational model and are displayed on the screen. The user defines the value correspondences by drawing lines from source to target. Given the defined value correspondences, the system computes, at a semantically complete mappings and presents it on the screen. Clio provides two ways for viewing a mapping. One is through values of the target elements. Next to each element of the target schema, the system displays how the value of the specific element will be computed by the mapping. If, for example, the element is the end point of a value correspondence, the element will contain the name of the source schema element at the other end of the value correspondence. Other elements may have no value at all, null, or a value produced by a Skolem function call. Since sometimes

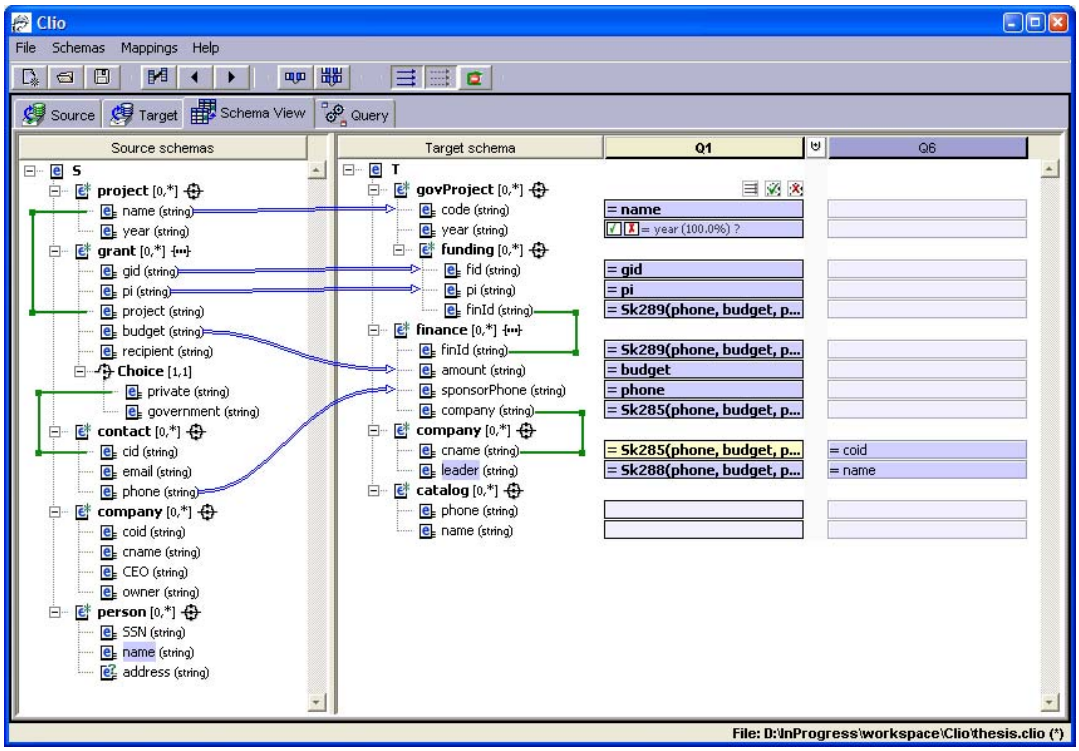


Figure 4.14: The graphical user interface of Clio.

this method may not provide all the details of the mapping, the second presentation method for a mapping can be used, which is the translation queries generated from the mapping (e.g., SQL query or XQuery). If the computed mapping does not reflect the intended semantics of the value correspondences, the system can be instructed to compute another semantically complete mapping. That way the user can browses through the mappings in the mapping universe until the result reflects the intentions of the user. At any time, the state of the system (schemas, value correspondences, mappings, etc.) can be saved to a file (Mapping Definition File - MDF) and later reloaded so that the mapping task can continue from the point at which it had been stopped.

- **Data Model Wrapper:** The role of this wrapper is to convert the schemas of the source and target databases to the internal nested relational model. One of the main goals of the module is to try to eliminate the peculiarities of the various

schema formalisms. In XML Schema, for example, a type name can be used even if the type is defined later in the XML Schema file. This is not the case for our model. Thus, the wrapper will try to detect those cases and make the appropriate expansion, i.e., replacing the type reference by its actual definition. The module uses JDBC[3] to communicate with relational databases and retrieve their catalog information. XML Schemas or DTDs can be loaded either from a local file or from the Web through the HTTP protocol. For parsing the latter two kinds of schemas, the XML4J[4] parser is used. The nested relational schema generated by this module is given to the *Context Manager* for type checking and maintenance.

- **Context Manager:** The *Context Manager* can be seen as the general coordinator of the system. Most of the commands go through this module. Its main role is to keep in memory the schemas and the value correspondences. It is also the static type checker of the system that ensures that schemas and value correspondences are well-defined. For the schemas, for example, it checks whether all the attributes of the record types have names, or whenever a type name is used, it makes sure that the type has already been defined. The types of the two schemas along with their mappings are stored in a special structure called *context*. The value correspondences are stored separately as a list, thus, value correspondences can easily be added or removed by appending or deleting elements to or from that list. The context and the list of value correspondences are given to the *Chase Engine*. Currently, Clio does not allow modifications to the context, i.e., the user cannot change the structure of the schemas. Chapter 5 will deal with this issue and will present a framework for dealing with schema changes and its implementation in a tool called ToMAS. New source schemas can be loaded and appended to those already loaded. Whenever such a context extension takes place or there is a modification in

[3]http://java.sun.com/products/jdbc/
[4]http://www.alphaworks.ibm.com/tech/xml4j

the value correspondence list, only the modifications are propagated to the *Chase Engine* and only the new parts are type checked.

- **Chase Engine:** The *Chase Engine* is responsible for performing the chase step. Given the context, it finds all the structural associations of the schemas. Then, it chases them using the schema constraints that are in the context, and generates all the source and target logical associations. Next, it produces a list of all the source-target logical association pairs. For each such pair, it checks what value correspondences are covered and stores them in a list associated with the specific pair. If a value correspondence is covered in more than one way, then each way is stored in the list. Pairs that cover no value correspondences are not eliminated since they may become useful for correspondences that are defined later. That way, redundant re-computations of pairs and coverages are avoided. When new schemas are appended to the context, only the pairs that are related with the new part of the schema are computed and the value correspondences are checked for coverage only against the new pairs. The list of pairs and the value correspondences they cover are given to the *Mapping Generator*.

- **Mapping Generator:** This module generates the mappings. For each pair of source and target logical associations that covers at least one value correspondence, one mapping is generated. If the value correspondence is covered by a pair in more than one way, then only one of them is displayed to the user. If the user is not satisfied with the mappings generated by the system, she can instruct the *Mapping Generator* through the GUI to use a different interpretation of the value correspondences, i.e., to consider different ways of coverage of the correspondences. The *Mapping Generator* will search for value correspondences that are covered in more that one way, select a different coverage, and recalculate the mappings.

- **Query Generator:** This module implements the query generation algorithm. It

generates the set ids and performs the appropriate grouping conditions. Among the mappings that were given by the *Mapping Generator*, it selects only a subset, and provides it to the GUI and therefore to the user. If no mapping in this subset reflects the intended semantics, *Query Generator* can be instructed to provide a larger subset. The selected set of mappings can then be given to one of the query wrappers (XQuery or SQL wrapper).

- **Query Wrappers:** *Query Wrappers* will take the mappings given by the *Query Generator* and turn them into queries in an actual query language (SQL, XQuery, or XSLT). *Query Wrappers* are the only modules of the system that deal with the specifics of the source query language. The queries produced by those modules can be sent either to the GUI for presentation and verification by the user, or to the data sources for execution.

4.6.2 Experimentation

Our experience has shown that most of the time the intended semantics of the correspondences are captured by the system, and the user effort in creating (and debugging) the translation queries is significantly less than with other approaches (including manually writing the query or using other query building tools). By user effort, we mean the tasks a user needs to perform and the decisions she might have to take. Although there is no standard benchmark or evaluation methodology for the subjective task of integration, we attempt to provide some evidence of the performance and effectiveness of our tool by discussing our usage of Clio on several schemas of different sizes and complexity. The tests were conducted by us, that were not the real users. However, the fact that we used schemas from well-known areas, e.g., bibliographic databases, allowed us to be able to evaluate the behavior of the system. In addition, the Clio tool has also been used by real users from the area of life sciences on large life science schemas. The anecdotal feedback

we received from these users was very encouraging since the users were happy with the tool. Clio helped them reduce the time they were spending in generating mappings, and they reported that in only few cases the results generated by Clio were not those desired. Although we did not perform any formal study on the reduction of the user time spent on generating mappings, the feedback we received indicates that even only the fact that the users are not required to manually write the mappings, but only graphically specify them in order to be generated automatically, is of a great benefit.

Our test schemas are listed in Table 4.1 with pairs of source and target schemas listed consecutively. They include: two XML Schemas for the DBLP bibliography (the first obtained from the DBLP Web Site)[5]; the relational `TPC-H`[6] schema and a nested XML view of this benchmark; two relational schemas from the `Amalgam`[7] integration suite for bibliographic data; the relational and DTD version of the `Mondial` database[8]; and two schemas representing a variety of gene expression (*micro-array*) experimental results created by the National Center for Genome Resources[9].

We included the DBLP schemas as examples of schemas with few constraints. These schemas still differ semantically, but the semantics is encoded primarily in the (different) nesting structure of the schemas, rather than through constraints. We wished to understand the effectiveness of our techniques in reconciling such differences. The DBLP_1 schema is an XML schema generated by the XML Spy tool (which automatically generates a schema from an XML document). It contains bibliographic citations organized by type (in-proceedings, articles, books, etc.). Within each publication type, there is a set of publications, each with a set of authors. In $\texttt{DBLP}_2$, the same information is rearranged by author. For each author, there is a set of publication forums, and for each forum, a set of years, then a nested set of articles. Moreover, $\texttt{DBLP}_2$ contains an additional view in

[5]http://dblp.uni-trier.de/db/
[6]http://www.tpc.org/tpch
[7]http://www.cs.toronto.edu/∼miller/amalgam
[8]http://www.informatik.uni-freiburg.de/∼may/Mondial/
[9]http://www.ncgr.org/genex

Schemas	Nest. Depth	Total Nodes	Leaf Nodes	NRIs	Load Time	Compile Time (sec)
$DBLP_1$ (XML)	2	88	52	0	0.52	0.19
$DBLP_2$ (XML)	4	27	12	1	0.15	0.15
TPC-H (RDB)	1	51	34	9	0.21	0.44
TPC-H (XML)	3	19	10	1	0.03	0.02
GeneX (RDB)	1	84	65	9	0.11	0.72
GeneX (XML)	3	88	63	3	0.13	0.50
Mondial (RDB)	1	159	102	15	0.58	5.41
Mondial (XML)	4	144	90	21	0.57	3.68
$Amalgam_2$ (RDB)	1	108	53	26	0.59	6.37
$Amalgam_1$ (RDB)	1	132	101	14	0.47	1.85

Table 4.1: Test schemas characteristics

which each author first appears nested by publication year and then by forum. To avoid redundancy, this view does not contain the actual article information, but rather refers to the article information (using a foreign key) in the first view of $\texttt{DBLP}_2$.

The `Amalgam` relational schemas, on the other hand, are examples of schemas without any nesting structure where all the semantics are captured by a rich set of referential constraints. The second `Amalgam` schema represents an extreme example of an overly normalized schema with many referential constraints. We included these schemas to illustrate the performance of our algorithms.

The remaining schemas use both constraints and nesting to represent semantics. We have also included real XML schemas with relatively few constraints (GeneX) and with many constraints (Mondial). In practice, we have encountered schemas of both types. For other schemas (notably the TPC-H and Mondial schemas), the XML or DTD was created to express the same data as the relational version. Hence, there is little heterogeneity in data content, but there was heterogeneity in the schemas' semantics, particularly in the nesting structure. Even for these relatively homogeneous schemas, it is nontrivial to manually write a query that correctly groups and nests the data into the XML output.

Table 4.1 shows some characteristics of all these schemas in our internal nested relational representation. The nesting depth indicates the nesting of repeated elements

Schema	Correspondences	With NRIs	No NRIs	Subsumed
DBLP	10	13	11	2
TPC-H	8	14	7	3
Amalgam	25	10	9	8
Mondial	43	5	4	2
Expense/StatDB	8	5	5	2
GeneX	30	4	4	1

Table 4.2: Source-to-target mappings generated with and without NRIs in the schemas

(set types). There is often more nesting through record types, but this does not affect the efficiency of our algorithms. The load and compile time indicate, respectively, the time to read the schemas and the time to bring the schemas into our internal mapping representation. The compile time includes the time to understand the nested structure (by computing structural associations) and to combine structures linked by constraints (using the chase to compute logical associations). While the load time, as expected, closely reflects the schema size (including both the schema structure and constraints), the compile time is mostly affected by the number of NRIs. For schemas with few or no NRIs, the compile time is almost negligible, while for schemas with a large number of NRIs, the time to compile is longer. Although compile time can be as large as several seconds (in our unoptimized prototype) for schemas with many constraints, we found this to be an acceptable delay for Clio users. Recall that compilation occurs only *once* when a schema is loaded.

To evaluate the results of our semantic translation, we sought to understand whether our algorithms were producing the right results and whether they were doing so in an effective way. Table 4.2 shows the total number of source-to-target mappings that Clio generates for our test schemas (column labeled "With NRIs"). Note that, since the experiments were done on schemas of various sizes, the number of correspondences was also different for each pair of schemas. For example, the number of correspondences in the experiment with the TPC-H schema was much smaller than the number of correspondences in the experiment with the Mondial schema, which is much larger. The column

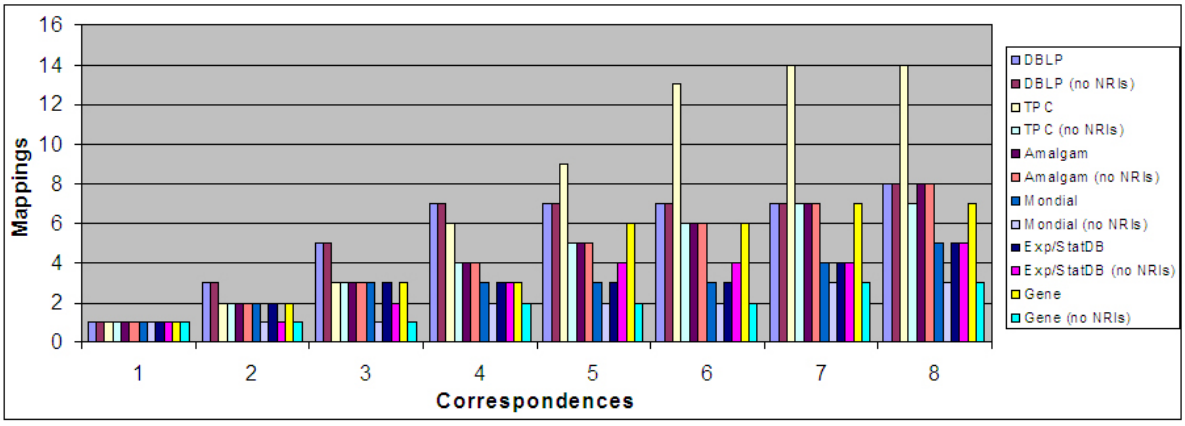


Figure 4.15: Change of the number of mappings as a result of addition of new correspondences.

labeled "Correspondences" in Table 4.2 indicates the number of correspondences that were used in the experiment that is shown in the table. For comparison to the number of generated mappings, we have also included the number of mappings that would have been produced if we had ignored all (intra-schema) constraints (column labeled "No NRIs"). These mappings preserve the semantics of nesting but ignore any referential semantics embedded in schema constraints. Although schema constraints may increase the number of possible mappings, we find that in practice the number of mappings remains manageable, and users do not get overwhelmed with too many choices. Furthermore, the last column (labeled "Subsumed") shows how many of the mappings generated without NRIs were subsumed by better, association-preserving mappings when NRIs were used. For instance, in the case of the `expenseDB/statDB` simple example, the number of mappings created with and without mappings is the same: five. Three of the mappings created without NRIs appeared again, unchanged, when NRIs were used. The other two mappings were subsumed by mappings that included the available NRIs. We, thus, end up with a better quality mapping that maintains the implied relationships in the schemas.

The chart in Figure 4.15 indicates how the number of mappings generated by the Clio tool changes as a result of the addition of new correspondences.

For all the schemas, the user was able to select a subset of the generated mappings to form the correct (desired) translation. So our algorithms were comprehensive enough to capture the heterogeneous semantics of all of these schemas. In some cases (e.g., `DBLP`, `Amalgam`, `expenseDB`), the intended transformation required all the created mappings, in other cases it required only a strict subset. Hence, we have chosen to not apply any heuristics to prune the complete set of mappings that Clio produces. Clio's DataViewer [YMHF01] has proven to be very effective in helping users understand and select the desired mappings.

Although the pairs of schemas we tested were describing similar information, Clio still needed to generate data values for each of our targets. For the `DBLP` target schema, which contained different nesting views of the same publications, we had to create values for article ids (via one Skolem function) to guarantee the consistency of the target data. In `TPC-H`, we required queries with five distinct Skolem functions, with some Skolem functions depending on as many as four different values. Correctly generating such a query by hand is an error-prone and complex task.

Clio is not the only tool that is currently available in the market. Commercial systems have numerous tools that are designed for data administrators in an effort to reduce the time and human effort that is required for the data translation task. Some of the tools available are the Aldove MapForce[10], the Tibco XML Transform[11], and the Data Junction[12]. However, none of these tools has the sophisticated reasoning of Clio. They have an interface similar to Clio, but cannot infer whether two correspondences should be considered together or not. Thus, a lot of human effort is still required for a data administrator to achieve the desired mapping.

The experiments we have conducted have shown that considering schema structure and constraints during mapping generation is a critical part of the process, since a signif-

[10]http://www.altova.com/products_mapforce.html
[11]http://www.tibco.com/software/business_integration/xmltransform.jsp
[12]http://www.pervasive.com/downloads/migrationtoolkit.asp

icant number of mappings would never have been generated if we had not. Furthermore, the complexity of the generated mappings indicate that using correspondences for the high level specification of how the elements of the schemas relate each other, is of great benefit to the user, since they are much simpler. Finally, the real schema we have used, showed that in practice, many schema elements may be left without their value specified by a mapping, hence, generating values for such element is important. Finally, our market research revealed that to the best of our knowledge, there are no commercially available tools that can generate queries at the level of complexity of Clio. Clio technology has already been incorporated into part of the IBM DB2 universal database.

Chapter 5

Mapping Maintenance

In many data management tools schemas are considered to be static. Yet, the Web is a dynamic environment with no centralized authority, and as often happens in heterogeneous dynamic environments, sources may evolve, without prior notice not only their context, but also their schemas or their query capabilities. Such changes must be reflected in the mappings since mappings necessarily depend on the schemas they relate. When a mapping is left inconsistent by a schema change, the inconsistency has to be detected and the mapping updated. As large, complicated schemas become more prevalent, and as data is reused in more applications, manually maintaining mappings (even simple mappings like view definitions) is becoming impractical and there is a need for automating this process. Most of the research efforts so far have considered only data changes. At the schema and the mapping level, the challenge is to realize the semantics of the schema change, and modify the mappings appropriately. This chapter presents such a framework, where the schema information in concert with the existing mappings are used to provide the right guidance for the mapping adaptation process. The chapter starts with a section introducing the problem of mapping maintenance and providing the necessary motivations (Section 5.1). Section 5.2 describes the algorithm for each possible kind of change that we can handle. The changes that we can handled. Our approach

considers not only local changes to a schema (Section 5.2.2), but also changes that may affect and transform many components of the schema structures (Section 5.2.3), and changes to the semantics of the schema, like changes to the schema constraints (Section 5.2.1). The mappings considered are from the expressive class of NRI constraints. The algorithm is complete in the sense that it detects mappings affected by structural and constraint changes and generates all the rewritings that are consistent with the semantics of the changed schemas. The approach explicitly models mapping choices made by a user and maintains these choices, whenever possible, as the schemas and mappings evolve. When there is more than one candidate rewriting, the algorithm may rank them based on how close they are to the semantics of the existing mappings, as well as previous mapping choices that the user has made (Section 5.3). An implementation of a mapping management and adaptation tool, called ToMAS, based on these ideas is described (Section 5.5.1) and is compared to the mapping generation tool Clio that was presented in the previous chapter (Section 5.5.2). In addition, a case study is presented using ToMAS to manage mappings in a physical database design scenario (Section 5.5.3).

The material of this Chapter has been published in VLDB 2003 [VMP03b] and VLDB Journal [VMP04]. The described implementation has been demonstrated in ICDE 2004 [VMPM04].

5.1 The Mapping Adaptation Problem

In dynamic environments like the Web, where there is no centralized authority, data sources may change not only their data, but also their schema, their semantics and their query capabilities. For instance, two tables may be merged into one, or attributes may be deleted. When such a modification takes place, some of the existing mappings may be left invalid or inconsistent, since the schema structures on which they were defined have changed. Those mappings have to be detected and modified accordingly.

One very common way to deal with this problem is to rely on a data administrator who has very good knowledge of the schema structure and semantics as well as the transformation language that is used. The data administrator goes through the defined mappings and for each one, checks whether the mapping is affected by the schema change that has taken place. If she finds one that is affected, she makes the necessary changes and adapts the mapping so that it is consistent with the new modified schema(s). In this process, the data administrator uses her knowledge and expertise to determine the most appropriate rewriting of the mapping. This process is time-consuming and error-prone. It is time-consuming since the data administrator has to go through numerous mappings, that describe complex and difficult to understand translations, and modify them. It is also error-prone since the data administrator needs to use her own knowledge, and there are no guarantees that the rewriting that results is the most appropriate one.

A second approach is to ignore the old mappings and redefine them from scratch for the new schemas. This approach can be aided tremendously by mapping generation tools, like Clio, which was presented in the previous chapter. This approach requires less effort from the data administrator, but also requires the regeneration of all the mappings, even those that were unaffected by the schema change. Furthermore, even with a mapping generation tool, some input from a data administrator is required to specify what mappings among all those generated by the mapping tool describe the exact semantics of the intended transformation. We need, therefore, a more automatic and incremental way to update the mappings affected by the changes, and preserve design choices that have been made by data administrators in the past. These choices are encoded in the mappings. The current chapter will present such a framework and its implementation. The resulting tool systematically detects mappings affected by the schema changes and rewrites them. During this process, the algorithm attempts to preserve the initial semantics of the affected mappings, as much as possible. For example, if two tables were joined in a specific way in a mapping, then unless this join makes

the mapping syntactically incorrect or semantically inconsistent, this way of joining is preserved. This problem of finding rewritings of mappings when schemas change is a new problem that we refer to as the *mapping adaptation problem* to differentiate it from the related, but different, problems of schema evolution [Ler00], view adaptation [GMR95], view synchronization [LNR02], and view maintenance [Wid95]. These research areas were presented in detail in Section 2.5.

One way to approach this problem is to have a predefined finite set of interesting changes. Indeed, this is the approach used in several application areas, like schema evolution, where for each change, the action that needs to be performed is stored ("hard-coded" if you will) in the application. The advantage of this approach is that it is known in advance how exactly each change is to be handled. The disadvantage is that the way in which the schemas can evolve is restricted to a set of predefined changes. However, if the set is rich enough, it may embrace all the possible schema changes that are important for a specific application. The other alternative, as mentioned before, is to allow schemas to evolve and then find the changes that took place by comparing the modified schema to its original version. For example, one could use a matching tool to find corresponding portions of the two schema versions [RB01] and then use a mapping creation tool to add semantics to these correspondences [PVM+02b]. This will produce a mapping from the new version of the schema to its old version. This mapping can then be composed with the initial mappings from the source to the target schema to generate the adapted mappings. This is the approach followed by the Rondo model management system [MRB03] and is graphically explained in Figure 5.1. Such an approach can be aided by the recent results in mapping composition [MH03, FKPT04]. However, the work on mapping composition is in its infancy. Madhavan and Halevy show that composition of mappings is feasible for mappings that can be written as a set of datalog rules, each one containing at most k variables, and the composition of such mappings is a finite set of datalog rules [MH03]. Furthermore, the semantics of mapping composition they defined is based on the certain

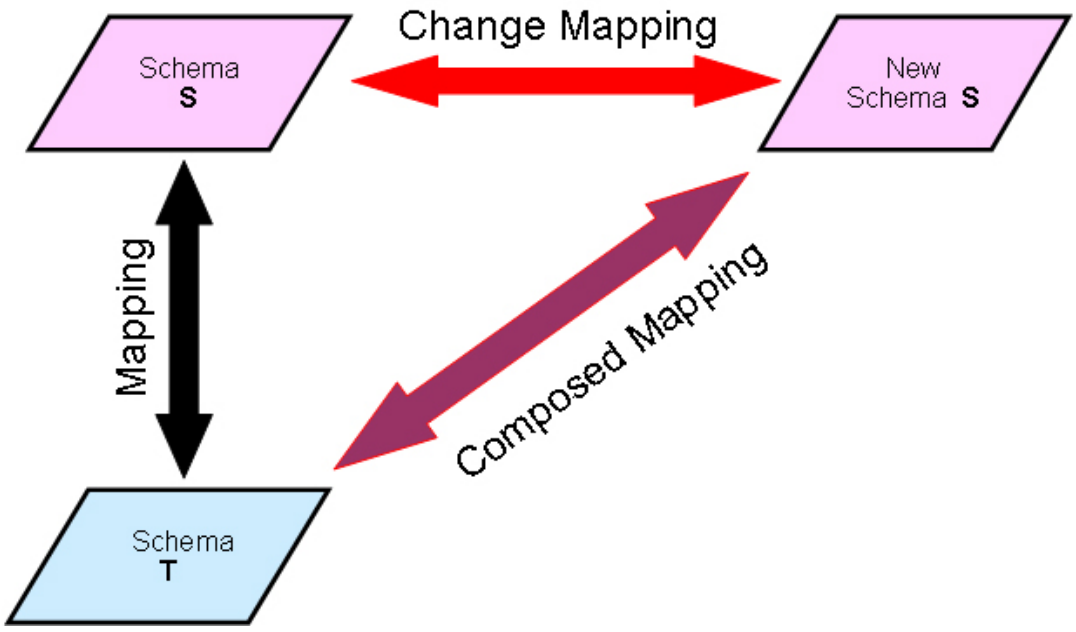


Figure 5.1: Model management approach to mapping maintenance.

answers of a query, which means that the composition process depends on the query we have at hand each time. On the other hand, Fagin et al. [FKPT04] redefined the mapping composition problem so that it is independent of any query and they show that in certain cases, the composed mapping may not be expressible using conjunctive queries even with an infinite set of datalog rules. Another issue that needs to be taken into consideration is that mapping composition may lead to semantic cluttering. The following example illustrates this. Consider a schema $\mathcal{S}$ consisting of relations $R_1(a, b, c)$, $R_2(g, h, d, e)$ and $R_3(w, f)$, where attributes g, h and w are references to attributes c, b, and e, respectively. Consider also a target schema $\mathcal{T}$ with the single relation $M(a, d, f)$, and the mapping from $\mathcal{S}$ to $\mathcal{T}$:

foreach $R_1\ s_1,\ R_2\ s_2,\ R_3\ s_3$
where $s_3.w{=}s_2.e$ and $s_2.g{=}s_1.c$
exists $M\ t$
with $t.a{=}s_1.a$ and $t.d{=}s_2.d$ and $t.d{=}s_3.f$

Assume that relation R_3 is removed from schema $\mathcal{S}$. The new version of the schema $\mathcal{S}$ will be the one with the two relations $R_1^n(a, b, c)$ and $R_2^n(g, h, d, e)$. Looking at the old and

the new version of the schema $\mathcal{S}$, and with the obvious correspondences between their attributes (i.e., a to a, b to b, etc.), the following mappings are semantically complete mappings that a mapping generation tool would have generated:

1: **foreach** $R_1^n\ s_1^n,\ R_2^n\ s_2^n$
where $s_2^n.g{=}s_1^n.c$
exists $R_1\ s_1,\ R_2\ s_2,\ R_3\ s_3$
where $s_3.w{=}s_2.e$ and $s_2.g{=}s_1.c$
with $s_1.a{=}s_1^n.a$ and $s_1.b{=}s_1^n.b$ and $s_1.c{=}s_1^n.c$
$s_2.g{=}s_2^n.g$ and $s_2.h{=}s_2^n.h$ and $s_2.d{=}s_2^n.d$ and $s_2.e{=}s_2^n.e$

2: **foreach** $R_1^n\ s_1^n,\ R_2^n\ s_2^n$
where $s_2^n.h{=}s_1^n.b$
exists $R_1\ s_1,\ R_2\ s_2,\ R_3\ s_3$
where $s_3.w{=}s_2.e$ and $s_2.g{=}s_1.c$
with $s_1.a{=}s_1^n.a$ and $s_1.b{=}s_1^n.b$ and $s_1.c{=}s_1^n.c$
$s_2.g{=}s_2^n.g$ and $s_2.h{=}s_2^n.h$ and $s_2.d{=}s_2^n.d$ and $s_2.e{=}s_2^n.e$

3: **foreach** $R_1^n\ s_1^n,\ R_2^n\ s_2^n$
where $s_2^n.g{=}s_1^n.c$
exists $R_1\ s_1,\ R_2\ s_2,\ R_3\ s_3$
where $s_3.w{=}s_2.e$ and $s_2.h{=}s_1.b$
with $s_1.a{=}s_1^n.a$ and $s_1.b{=}s_1^n.b$ and $s_1.c{=}s_1^n.c$
$s_2.g{=}s_2^n.g$ and $s_2.h{=}s_2^n.h$ and $s_2.d{=}s_2^n.d$ and $s_2.e{=}s_2^n.e$

4: **foreach** $R_1^n\ s_1^n,\ R_2^n\ s_2^n$
where $s_2^n.h{=}s_1^n.b$
exists $R_1\ s_1,\ R_2\ s_2,\ R_3\ s_3$
where $s_3.w{=}s_2.e$ and $s_2.h{=}s_1.b$
with $s_1.a{=}s_1^n.a$ and $s_1.b{=}s_1^n.b$ and $s_1.c{=}s_1^n.c$
$s_2.g{=}s_2^n.g$ and $s_2.h{=}s_2^n.h$ and $s_2.d{=}s_2^n.d$ and $s_2.e{=}s_2^n.e$

Which of these mappings express the right transformation cannot be inferred automatically by looking at the old and the new version of the schema $\mathcal{S}$ alone. An expert user is needed to do this selection. If not, each one of these mappings has to be considered in combination with the initial mapping from $\mathcal{S}$ to $\mathcal{T}$. Even though, mappings 2 and 3 turn out to be equivalent, i.e., one is logically implied by the other, the composition with

the initial mapping will result in more than one mapping from $\mathcal{S}^n$ to $\mathcal{T}$. On the contrary, with the incremental approach, only one mapping would have been created from $\mathcal{S}^n$ to $\mathcal{T}$:

foreach $R_1^n\ s_1^n,\ R_2^n\ s_2^n$
where $s_2^n.g{=}s_1^n.c$
exists $M\ t$
with $t.a{=}s_1.a$ and $t.d{=}s_2.d$ and $t.d{=}s_3.f$

Despite its difficulties, the composition approach is inevitable in cases where the sequence of the changes is not available, but only the initial and final versions of the schemas. To this end, a number of methods [CGM97, LGM96] have been developed trying to compare two structures and identify the changes that led from the first to the second.

The approach presented in this chapter is to use a mapping adaptation tool with which a designer can change and evolve schemas. The tool detects mappings that are made inconsistent by a schema change and allows the mappings to be incrementally modified in response. The term *incrementally* means that only the mappings and, more specifically, the parts of the mappings, that are affected by a schema change are modified, while the other parts remain unaffected. As mentioned before, this approach has the advantage that one can track semantic decisions made by a designer either in creating the mappings or in earlier modification decisions. These semantic decisions are needed because schemas are often ambiguous (or semantically impoverished) and may not contain sufficient information to make all mapping choices. One can then reuse these decisions when appropriate. The main idea for the mapping adaptation process is the following. For each change that takes place in the source or in the target schema, the mappings that are affected are identified and the parts that need modification are determined. By performing the minimum required computations, these mappings are updated. Based on the schema change, there may be a number of candidate rewritings for a mapping. The basic criterion for selecting the best rewriting is semantic preservation. For each

rewriting, the tool measures how semantically close the new mapping (the one after the rewriting) is to the initial mappings (the one that was affected by the schema change), and the one that is closest is selected. One would expect a procedure like this to generate rewritings that are equivalent to the original mapping. Unfortunately, this may not always be possible since, when a schema changes, inevitably, the schema semantics also change. Thus, the requirement for finding only equivalent rewritings is too strict and must be relaxed.

The main contributions of this chapter are the following.

1. The problem of adapting mappings to schema changes is motivated and a simple and powerful model for representing schema changes is presented.

2. The changes that are considered are not only to the structure of schemas (which may make the mapping syntactically incorrect [BHL83]) but also to the schema semantics that may make mappings semantically incorrect.

3. An algorithm for enumerating possible rewritings for mappings that have become invalid or inconsistent is presented. The generated rewritings are consistent not only with the structure but also with the semantics of the schema.

4. A method that provides an indication for measuring the semantic similarity between two mappings is defined and used to rank the candidate rewritings.

5. The changes that are considered may be not only in the source schemas but also in the target. This is equivalent to adapting mappings to reflect changes in both their interface and the base schema.

6. Changes may be not only on atomic elements, but also on more complex structures including relational tables or complex (nested) XML structures.

7. A mapping adaptation algorithm that efficiently computes rewritings by exploiting

m_1: **foreach** S.projects p, S.grants g,
g.grant.sponsor→government r, S.contacts c,
where p.project.name=g.grant.project and
r=c.contact.cid
exists T.govProjects j, j.govProject.fundings u,
T.finances f, T.companies o
where u.funding.finId=f.finances.finId and
f.finances.company=o.company.cname
with j.govProject.code=p.project.name and
u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and
f.finances.amount=g.grant.budget and
f.finances.sponsorPhone=c.contact.phone

m_2: **foreach** S.companies w, S.persons n
where n.person.SSN=w.company.owner
exists T.companies o
with o.company.cname=w.company.cname and
o.company.leader=n.person.name

m_3: **foreach** S.contacts c, S.persons k
where k.person.SSN=c.contact.cid
exists T.catalog a
with e.entry.name=k.person.name and
e.entry.phone=c.contact.phone

Figure 5.2: Three mappings that form with the schemas of Figure 3.1 a mapping system.

knowledge about user decisions that are embodied in the existing mappings is presented.

5.2 Schema Evolution

When schemas evolve to adapt to new data requirements, mappings should be rewritten, not only to continue to be valid, but also to become consistent with the semantics of the new schema, while at the same time preserving as much as possible the intended initial transformation. This sections provides the algorithms for handling schema evolution

that achieve these two goals. In particular, to achieve the former, we exploit information provided by the schema structure and semantics (constraints). We provide here an algorithm to efficiently compute the schema semantics incrementally when a change to the schema structure or constraints occurs. For the latter, we present new techniques for modeling and reusing the semantics embedded within a mapping.

When schemas evolve, they usually do not change radically. In most practical cases, the changes are small restructuring changes that are performed either to improve performance or to reflect new data requirements. In these cases, the evolution of a schema can be expressed as a sequence of primitive changes, which makes our approach useful and applicable. This chapter identifies a set of primitive changes of different kinds and explains how mappings should be modified when each such change takes place. For each change, it describes how to detect the mappings that are affected by the change and how to realize the rewriting that needs to be performed.

The mapping adaptation procedure needs as input a mapping system, i.e, a pair of schemas $\mathcal{S}$, $\mathcal{T}$ and a set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$. It consists of two phases. The first is a preprocessing step in which the mappings are analyzed and turned into semantically complete mappings (if they are not). In particular, the set $\mathcal{C}$ of correspondences covered by the mappings in $\mathcal{M}$ is first extracted and then the mappings in $\mathcal{M}$ are analyzed. For each mapping $m \in \mathcal{M}$ of the form **foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D, associations $A^{\mathcal{S}}$ and $A^{\mathcal{T}}$ are taken apart and are chased with the schema constraints to produce new associations $A_1^{\mathcal{S}}, ..., A_n^{\mathcal{S}}$ and $A_1^{\mathcal{T}}, ..., A_l^{\mathcal{T}}$ respectively. This brings in additional joins of which the user may not have been aware. For each pair $<A_i^{\mathcal{S}}, A_j^{\mathcal{T}}>$, a new mapping m_{ij} of the form **foreach** $A_i^{\mathcal{S}}$ **exists** $A_j^{\mathcal{T}}$ **with** D' is created, where D' includes the conditions D, plus the conditions of all the correspondences that are covered by the pair of associations $<A_i^{\mathcal{S}}, A_j^{\mathcal{T}}>$, but were not covered by the pair $<A^{\mathcal{S}}, A^{\mathcal{T}}>$. This is because associations $A^{\mathcal{S}}$ and $A^{\mathcal{T}}$ are always dominated by associations $A_i^{\mathcal{S}}$ and $A_j^{\mathcal{T}}$, respectively. Note that the set of all those semantically complete mappings m_{ij} is a subset of the mapping universe

$\mathcal{U}^{\mathcal{M}}_{\mathcal{S},\mathcal{T}}$. Any existing user association is also identified during this process.

The second phase of the process takes the set of semantically complete mappings generated during the first phase and maintains them through schema changes. In particular, for each kind of change that occurs in the source or the target schema, each mapping is modified appropriately. This is done for each mapping in turn. Note that mappings generated in the first phase are potentially more complete than those entered by a user. Hence, we use the generated mappings within the adaptation algorithm since they extend the user mappings with the semantics embedded in the schema structure and constraints. One fundamental reason for using the mappings generated in the first phase and not those that were initially given, besides the fact that they are more "complete" than what the user may have entered, is that the semantics that are embedded in the schema structure and constraints give a reasonably good idea of what the adapted mappings should be.

The mapping adaptation procedure is described by an algorithm that consists of a number of mutually exclusive parts. Each part handles a specific kind of change that may occur in the schema. The following subsections will present these parts, one after the other. Each part accepts as input a set of semantically complete mappings $\mathcal{M}$ and returns the set of adapted semantically complete mappings $\mathcal{M}'$. The primitive schema changes that we have identified can be classified in three categories:

1. Operations that change the schema semantics by adding or removing constraints,

2. Modifications to the schema structure by adding or removing elements, and

3. Changes that reshape the schema structure by moving, copying or renaming elements.

To make the presentation less verbose, we will often assume that the schema changes occur in the source schema. However, the algorithms apply equally to the case in which the changes occur in the target schema.

Given a schema mapping system O with a universe $\mathcal{U}$, every schema change modifies the universe $\mathcal{U}$ of O into a new universe $\mathcal{U}'$. A mapping $m \in \mathcal{U}$ survives the schema change, i.e., does not have to be adapted, if $m \in \mathcal{U}'$ as well; otherwise, m has to be replaced by a new mapping $m' \in \mathcal{U}'$ that was not in $\mathcal{U}$.

Example 5.1 *Consider the mapping scenario of Figure 3.1 and assume that the user has generated (either manually or through a mapping generation tool) the set of three mappings that are indicated in Figure 5.2. (The schemas of Figure 3.1 and the mappings of Figure 5.2 form a mapping system.) Mapping* m_1 *is a mapping that populates the target with projects that have government sponsors. Note that the* `company` *of* `finances` *is asserted to be equal to a company with name* `cname`*. However, the* **with** *clause does not relate any source elements with target element* `cname` *or* `company`*. So* m_1 *constrains the target instance data that can be generated, but does not completely specify the target instance. The mapping is a semantically complete mapping. The logical associations it is using are the associations* A_3^S *and* A_2^T *of Figure 4.3. If the mapping did not have the last variable bound to* `companies` *of the target schema, it would not have been semantically complete. In that case, the preprocessing step of our algorithm would have added this binding (and the respective join condition in the* where *clause) to make it semantically complete. Mapping* m_2 *is a mapping that specifies how to populate the target with companies consisting of the company name from the source and the name of the company owner as a leader. The third mapping* m_3 *generates catalog entries in the target by joining persons and contacts in the source. Looking at the associations in Figure 4.3, one can see that there is no logical association that joins* `contacts` *and* `persons`*. Hence, the specific mapping could not have been generated by the mapping generation algorithm presented in the previous chapter, which means that it has been specified by the user explicitly. As such, the association*

```
select *
from S.contacts c, S.persons k
```

where c.contact.cid=k.person.SSN

is a user association.

The two schemas of Figure 3.1 along with the three mappings of Figure 5.2 form a schema mapping system that we will use as an example in the following sections. ■

5.2.1 Constraint Modification

This section describes the first category of schema modifications we handle, which is changes to the schema constraints.

Adding Constraints

Adding a new constraint to a schema does not make any of the existing mappings invalid, i.e., syntactically incorrect. However, it may make some of the mappings inconsistent, in the sense that they will no longer reflect the intended semantics of the schema. More precisely, a mapping may fall out of the mapping universe as a result of adding a constraint. Let $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ be a mapping system, and $\mathcal{C}$ be the set of correspondences dominated by all the mappings in $\mathcal{M}$ (which can always be extracted from the set of mappings $\mathcal{M}$, as mentioned in previous section).

Assume that a new constraint F: **foreach** X **exists** Y **with** C is added in the source schema. We first detect the mappings that are affected by the change, that is, mappings that are not semantically valid any more according to the new constraint. A mapping m:**foreach** A^S **exists** A^T **with** D, with $m \in \mathcal{M}$ of a mapping system $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ needs to be adapted after the addition of a source constraint **foreach** X **exists** Y **with** C if X is dominated by $A^{\mathcal{S}}$ ($X \dot{\preceq} A^{\mathcal{S}}$), with a renaming h, but there is no extension of h to a renaming from $Y \sqcup C$ to $A^{\mathcal{S}}$. In other words, the addition of the new constraint causes $A^{\mathcal{S}}$ to not be closed under the chase, i.e., $A^{\mathcal{S}}$ is no longer a logical association.

The reason this is true is the following. The new constraint affects neither the structure of the schemas, nor the old constraints. If mapping m has to be adapted, it is because

the logical association $A^{\mathcal{S}}$ has to change. The only reason this can happen is because it can be chased with the new constraint F, which means that $X \dot{\preceq} A^{\mathcal{S}}$ and $Y \sqcup C \dot{\not\preceq} A^{\mathcal{S}}$. Contrapositively, if $X \dot{\preceq} A^{\mathcal{S}}$ and $Y \sqcup C \dot{\not\preceq} A^{\mathcal{S}}$, then $A^{\mathcal{S}}$ can be chased further with the new set of constraints and generate a set of new logical associations.

If mapping m: **foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D, with $m \in \mathcal{M}$ of the mapping system $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$ needs to be adapted, the association A^S is chased with the set of old schema constraints augmented with the new constraint F. Note that it is not enough to chase only with F. After applying a chase step with F, additional chasing may be possible with existing constraints. The result is a set of new logical associations. For each such association A, a new mapping is generated of the form m_c: **foreach** A **exists** $A^{\mathcal{T}}$ **with** D'. The set D' consists of the conditions derived from the **with** clause of the correspondences in $\mathcal{C}$ that are covered by the pair $<A, A^{\mathcal{T}}>$. Since $A^{\mathcal{S}}$ is always dominated by A, naturally, $D \subseteq D'$. Each mapping m_c generated by the above procedure for which $m \dot{\preceq} m_c$ is added to $\mathcal{M}$, and mapping m is removed. Algorithm 3 gives a brief description of the steps taken when a new constraint is added.

Example 5.2 *Assume that the following new constraint is added in the source schema of Figure 3.1:*

f_8: **foreach** S.grants g
exists S.companies c
with c.company.cname $=$ g.grant.recipient

allowing each grant to specify the company that receives the grant. Mappings m_2 and m_3 will not be affected since the **foreach** *part of the constraint is not dominated by the* **foreach** *part of those mappings. Indeed, the fact that we can now determine the company that receives each grant has nothing to do with those two mappings since they deal with companies and persons only. However, this change greatly affects mapping m_1. Remember that the specific mapping was populating the target schema with projects receiving government funds and the associated companies, but the information about what*

Algorithm 3: Constraint Addition Mapping Adaptation Algorithm

Input: A source schema $\mathcal{S}$
A target schema $\mathcal{T}$
A set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$
A constraint F: **foreach** X **exists** Y **with** C of schema $\mathcal{S}$
Output: A new set of mappings $\mathcal{M}'$

ADDCONSTRAINT($\mathcal{S}$,$\mathcal{T}$,$\mathcal{M}$,F)
(1) $\mathcal{X} \leftarrow$ constraints in $\mathcal{S}$
(2) $\mathcal{M}' \leftarrow \emptyset$
(3) $\mathcal{C} \leftarrow$ compute correspondences from $\mathcal{M}$
(4)
(5) **foreach** $m \leftarrow$(**foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D) $\in \mathcal{M}$
(6) **if** $X \dot{\preceq} A^{\mathcal{S}}$ with renaming h and $h(Y \sqcup C) \dot{\not\preceq} A^{\mathcal{S}}$
(7) **foreach** $A \in chase_{\mathcal{X} \cup \{F\}}(A^{\mathcal{S}})$
(8) **foreach** coverage of $<A, A^{\mathcal{T}}>$ by $D' \subseteq \mathcal{C}$
(9) $m_r \leftarrow$ **foreach** A **exists** $A^{\mathcal{T}}$ **with** D'
(10) **if** $(m \dot{\preceq} m_r)$
(11) $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{m_r\}$
(12) **else**
(13) $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{m\}$
(14) **return** $\mathcal{M}'$

---

company is related to each project was not available in the source schema, hence, the mapping was not generating any `company` *subelement values in the target. After the addition of the new constraint, this information becomes available, so mapping* m_1 *may need to be adapted to the new schema semantics. We detect this by verifying that the* **foreach** *clause of the constraint is dominated by association* $A_3^{\mathcal{S}}$ *used in mapping* m_1 *through a renaming* h*, but there is no extension of* h *such that the union of the* **exists** *and* **with** *clauses of the mapping is also dominated by association* $A_3^{\mathcal{S}}$ *through that extension. When chased with the new set of constraints that includes* F*, association* $A_3^{\mathcal{S}}$ *gives a new logical association:*

$A_{3^n}^{\mathcal{S}}$: select *
from S.grants g,
g.grant.sponsor→government r,
S.projects p, S.contacts c

S.companies w, S.persons n, S.persons s
where p.project.name=g.grant.project and
c.contact.cid=r and
w.company.cname=g.grant.recipient and
w.company.CEO=s.person.SSN and
w.company.owner=n.person.SSN

The pair of associations $<A^{\mathcal{S}}_{3^n}, A^{\mathcal{T}}_2>$ *covers the correspondences* v_1, v_2, v_3, v_4 *and* v_5 *the same way they were covered by the pair* $<A^{\mathcal{S}}_3, A^{\mathcal{T}}_2>$. *In addition, it covers two new correspondences,* v_6 *and* v_7. *For* v_7 *specifically, which is :*

v_7: **foreach** S.persons d
exists T.companies i
with i.company.leader=d.person.name

there are two ways of coverage. The first is by mapping the variable d *to variable* n, *in which case the* leader *of a* company *is considered to be the* owner *of the company. The second is by mapping variable* d *to variable* s, *in which case the* leader *of a* company *is considered to be its* CEO. *Each different way of coverage generates in turn a different rewriting for the mapping* m_1. *The first rewriting (after some join minimization) becomes:*

m'_{1a}: **foreach** S.projects p, S.grants g,
g.grant.sponsor→government r, S.contacts c,
S.companies w, S.persons n
where p.project.name=g.grant.project and
r=c.contact.cid and
w.company.cname=g.grant.recipient and
w.company.owner=n.person.SSN
exists T.govProjects j, j.govProject.fundings u,
T.finances f, T.companies o
where u.funding.finId=f.finances.finId and
f.finances.company=o.company.cname
with j.govProject.code=p.project.name and
u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and
f.finances.amount=g.grant.budget and
f.finances.sponsorPhone=c.contact.phone and
o.company.cname=w.company.cname and
o.company.leader=n.person.name

The second way of coverage gives the following mapping rewriting, which differs only in the last line of the where *clause of the* **foreach** *part:*

m'_{1b}: **foreach** S.projects p, S.grants g,
g.grant.sponsor→government r, S.contacts c,
S.companies w, S.persons s
where p.project.name=g.grant.project and
r=c.contact.cid and
w.company.cname=g.grant.recipient and
w.company.CEO=s.person.SSN
exists T.govProjects j, j.govProject.fundings u,
T.finances f, T.companies o
where u.funding.finId=f.finances.finId and
f.finances.company=o.company.cname
with j.govProject.code=p.project.name and
u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and
f.finances.amount=g.grant.budget and
f.finances.sponsorPhone=c.contact.phone and
o.company.cname=w.company.cname and
o.company.leader=s.person.name

Both mappings are semantically complete mappings and describe similar (but different) mapping semantics. Deciding which one should be accepted as a rewriting cannot be done by a tool that simply looks at the schema structure and constraints. The involvement of a human is required. For that reason, the algorithm will consider both rewritings. ■

As the above example indicates, mapping adaptation may produce rewritings with different (maybe conflicting) semantics. In the absence of any additional information, it may not be possible to choose one mapping rewriting over another. All the generated rewritings are consistent with the new schema, and are semantically as close as possible to previously defined mappings (i.e., they are valid members of the new mapping universe). To accept or to reject some of them requires human intervention. In Section 5.3, we describe a methodology for ranking the generated rewritings based on their semantic

closeness to the previous mappings. This ranking can be used to assist the user in such a decision.

A special, yet interesting, case is when the chase does not introduce any new schema elements in the association but only extra conditions. These conditions indicate new ways (join paths actually) to relate the elements in the association. This enhances the mapping universe with new semantically valid mappings, however, none of the existing mappings is adapted. The intuition behind this is that there is no indication (from existing mappings or from schema structure and constraints) that these new mappings are preferred to those that already exist. Since our goal is to maintain the semantics of existing mappings as much as possible, we perform no adaptation without a reason.

Example 5.3 *Assume, for the sake of illustration, that a new foreign key constraint is added in the source schema of Figure 3.1 from the element* `budget` *to the element* `year`*. Mappings m_2 and m_3 are not affected by this change. Chasing association A_1 with the schema constraints introduces a new join path through which a* `project` *and a* `grant` *can be associated. However, there is no reason to assume that mapping m_1 needs to be adapted since the initial way of joining projects, grants and contacts (through the foreign key f_1) is still valid.* ∎

Removing Constraints

As with the addition of a constraint, the removal of a constraint has no effect on the validity of the existing mappings, but may affect the consistency of their semantics. The reason is that mappings may have used assumptions that were based on the constraint to be removed. As before, we assume that a source constraint is removed. (The same reasoning applies for the removal of a target constraint.) We consider a mapping to be affected if its source association uses some join condition(s) based on the constraint being removed. A mapping m will be affected by the removal of a constraint F only if it has used that constraint. Constraints are used in mappings through their logical

associations. Thus, to find the affected mappings, it is enough to find the affected logical associations. More precisely, a mapping m: **foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D, with $m \in \mathcal{M}$ of a mapping system $<\mathcal{S}, \mathcal{T}, \mathcal{M}>$, needs to be adapted after the removal of a source constraint F: **foreach** X **exists** Y **with** C if $X \sqcup Y \sqcup C \dot{\preceq} A^{\mathcal{S}}$. The reason is the following. If $X \sqcup Y \sqcup C \dot{\preceq} A^{\mathcal{S}}$ then it is natural that $X \dot{\preceq} A^{\mathcal{S}}$. Since $A^{\mathcal{S}}$ is a logical association, the constraint has been used during the chase. If it had not been used, the chase would not have been completed, and $A^{\mathcal{S}}$ would not have been a logical association. Hence, the removal of the constraint affects the form of the logical association, and as a consequence, the mapping. Contrapositively, if the association needs to be modified, it means that there is a step in which the constraint was used. The resulting association A of that chase step satisfies $X \sqcup Y \sqcup C \dot{\preceq} A$. Since the chase step only adds elements to the association but never removes any, it is true that $A \dot{\preceq} A^{\mathcal{S}}$, which means that it is also true that $X \sqcup Y \sqcup C \dot{\preceq} A^{\mathcal{S}}$.

Once we detect that a mapping m needs to be adapted, we apply the set of steps described by Algorithm 4. The intuition of Algorithm 4 is to identify and use the maximal independent sets of semantically associated schema elements of each affected association used by the mapping. We start by breaking apart the source association $A^{\mathcal{S}}$ into its set $\mathcal{P}$ of structural and user associations. That is, we enumerate all the structural and user associations of the source schema that are dominated by $A^{\mathcal{S}}$. We then chase them by considering the set of schema constraints *without* F. The result is a set of new logical associations. Some of them may include choices (due to the existence of choice types) that were not part of the original association $A^{\mathcal{S}}$. We eliminate such associations. The criterion is based on dominance again: we only keep those new logical associations that are dominated by $A^{\mathcal{S}}$. Let us call this set of resulting associations $\mathcal{A}'$. By construction, the logical associations in $\mathcal{A}'$ will contain only elements and equalities between them that were also in $A^{\mathcal{S}}$, hence, they will not represent any additional semantics. For every member A_a in $\mathcal{A}'$ and for every way A_a can be dominated by association $A^{\mathcal{S}}$ (i.e., for every

renaming function $h{:}A_a \rightarrow A^{\mathcal{S}}$), a new mapping m_a is generated of the form **foreach** A_a **exists** $A^{\mathcal{T}}$ **with** D'. The set D' consists of those correspondences that are covered by the pair $<A_a, A^{\mathcal{T}}>$ and such that their renaming h is included in D or implied by it. In other words, D' consists of the correspondences of D that are covered by $<A_a, A^{\mathcal{T}}>$ and $<A^S, A^{\mathcal{T}}>$ in the same way. Let us call M^* the set of mappings m_a. From the mappings in M^*, we wish to keep only those that are as close as possible to the initial mapping m. This is achieved by eliminating every mapping in M^* that is dominated by another mapping in M^*, so we keep only those that have the maximum number of elements. The following example illustrates the algorithm.

Example 5.4 *Consider the mappings m'_{1a}, m'_{1b}, m_2 and m_3 in Example 5.2 and let us remove the constraint f_8 we added there. Mappings m_2 and m_3 are not affected because they do not include a join between* `grants` *and* `companies`*. However, both m'_{1a} and m'_{1b} are affected. Consider the mapping m'_{1a} (m'_{1b} is handled in a similar way). Its source association, $A_3^{\mathcal{S}}$ of Example 5.2, is broken apart into the structural associations (recall Figure 4.2): $P_1^{\mathcal{S}}$, $P_3^{\mathcal{S}}$, $P_4^{\mathcal{S}}$, $P_5^{\mathcal{S}}$ and $P_6^{\mathcal{S}}$. Those structural associations are chased, and the result is a set of logical associations. $P_1^{\mathcal{S}}$ results in logical association $A_1^{\mathcal{S}}$ (recall Figure 4.3). $P_3^{\mathcal{S}}$ results in $A_3^{\mathcal{S}}$. The chase of $P_4^{\mathcal{S}}$, $P_5^{\mathcal{S}}$ and $P_6^{\mathcal{S}}$ will result in associations $A_4^{\mathcal{S}}$, $A_5^{\mathcal{S}}$ and $A_6^{\mathcal{S}}$ respectively. Each of the resulting associations will be used to form a new mapping. Association $A_1^{\mathcal{S}}$ covers only correspondence v_1, and generates the mapping:*

$m_{A_1^{\mathcal{S}}}$: **foreach** `S.projects` p
exists `S.govProjects` j
with j`.govProject.code` $= p$`.project.name`

Association $A_3^{\mathcal{S}}$ covers correspondence v_1 to v_5 and generates the mapping m_1 (which is the one we started with in Example 5.2). Association $A_4^{\mathcal{S}}$ covers only correspondence v_5, and generates mapping

$m_{A_4^{\mathcal{S}}}$: **foreach** `S.contacts` c
exists `S.finances` f
with f`.finances.sponsorPhone` $= c$`.contact.phone`

Association $A_5^{\mathcal{S}}$ covers correspondences v_6 and v_7. In particular, v_7 is covered in two ways. These two ways lead to two different mappings. The first is mapping m_2 of our mapping system (Figure 5.2), and the second is the mapping m_2 of Example 4.30. However, the second mapping is not dominated by mapping m'_{1a}, which we are adapting, since it covers the correspondence that originates on the `leader` *through a join on the* `CEO` *instead of the* `owner`*, hence it is eliminated. Finally, association $A_6^{\mathcal{S}}$ generates the mapping*

$$
\begin{array}{lll}
m_{A_6^{\mathcal{S}}}: & \underline{\textbf{foreach}} & \texttt{S.person}\ p \\
& \underline{\textbf{exists}} & \texttt{S.companies}\ o \\
& \underline{\textbf{with}} & o.\texttt{company.leader} = p.\texttt{person.name}
\end{array}
$$

Among the mappings $m_{A_1^{\mathcal{S}}}$, m_1, $m_{A_4^{\mathcal{S}}}$, m_2 and $m_{A_6^{\mathcal{S}}}$ that have just been generated, mappings $m_{A_1^{\mathcal{S}}}$ and $m_{A_4^{\mathcal{S}}}$ are dominated by mapping m_1 and are eliminated. Similarly, $m_{A_6^{\mathcal{S}}}$ is dominated by mapping m_2, and is also eliminated. The final result of the algorithm, for the case of m'_{1a}, consists of the mappings m_1 and m_2. The algorithm will give similar results for the case of m'_{1b}, and in particular, mapping m'_{1b} will be adapted to two mappings. The first is mapping m_1, as in the case of mapping m'_{1a}, and the second is the mapping m_2 but with the `leader` *of the* `company` *being the* `CEO` *instead of the* `owner`*. Mappings m_2 and m_3 are not affected by the removal of constraint f_8 and will not be modified. Consequently, the output of the algorithm will be four mappings, the three mappings of Figure 5.2, plus the one which is like m_2 but with a join on the* `CEO` *instead of the owner.*

■

The above example illustrates an interesting property of the mapping adaptation algorithm, that the order in which the changes are performed is not symmetric, i.e., performing a schema change x and then its reverse operation, may not result in the original mappings. Example 5.2 started with the mappings of Figure 5.2 and, by adding a constraint, made a new mapping system with four mappings. Then Example 5.4 removed that same constraint, and the result is not the one of Figure 5.2. The reason for

Algorithm 4: Constraint Removal Mapping Adaptation Algorithm

Input: A source schema $\mathcal{S}$
A target schema $\mathcal{T}$
A set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$
A constraint F: **foreach** X **exists** Y **with** C of schema $\mathcal{S}$
Output: A new set of mappings $\mathcal{M}'$

REMCONSTRAINT($\mathcal{S}$,$\mathcal{T}$,$\mathcal{M}$,F)
(1) $\mathcal{X} \leftarrow$ constraints in $\mathcal{S}$
(2) $\mathcal{M}' \leftarrow \emptyset$
(3) $\mathcal{C} \leftarrow$ compute correspondences from $\mathcal{M}$
(4) **foreach** m:(**foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D) $\in$ $\mathcal{M}$
(5) **if** $(X \sqcup Y \sqcup C \dot{\preceq} A^{\mathcal{S}})$
(6) $\mathcal{P} \leftarrow \{P \mid P$ structural or user association $\wedge\ P \dot{\preceq} A^{\mathcal{S}}\}$
(7) $\mathcal{A}' \leftarrow \{A \mid A \in chase_{\mathcal{X}-\{F\}}(P) \wedge P \in \mathcal{P} \wedge A \dot{\preceq} A^{\mathcal{S}}\}$
(8) $\mathcal{M}^* \leftarrow \emptyset$
(9) **foreach** $A_a \in \mathcal{A}'$
(10) **foreach** renaming h:$A_a \rightarrow A^S$
(11) $D' \leftarrow \{e_1 = e_2 \mid e_1(e_2)$ well defined expressions over
(12) A_a (A^T) $\wedge$ "$h(e_1)=e_2$" in or implied by D $\}$
(13) $m_a \leftarrow$ (**foreach** A_a **exists** A^T **with** D')
(14) $\mathcal{M}^* \leftarrow \mathcal{M}^* \cup \{m_a\}$
(15) $\mathcal{M}^{**} \leftarrow \{m' \mid m' \in \mathcal{M}^* \wedge \nexists\ m'' \in \mathcal{M}^* : m' \dot{\preceq} m''\}$
(16) $\mathcal{M}' \leftarrow \mathcal{M}' \cup \mathcal{M}^{**}$
(17) **else**
(18) $\mathcal{M}' \leftarrow \mathcal{M}' \cup \{m\}$
(19) **return** $\mathcal{M}'$

---

this is that the decisions of the algorithm are based always on the two schemas and the current set of mappings. So, the algorithm was based only on the set of four mappings from Example 5.2, and generated the mappings in Example 5.4.

5.2.2 Schema Pruning and Expansion

Among the most common changes that are made in schema evolution systems are those that add or remove parts of the schema structure, for example, adding a new attribute to a relational table or removing an XML Schema element. This section describes how

we can handle such cases.

When a new structure is added to a schema, it may introduce some new structural associations. Those structural associations can be chased in order to generate new logical associations. Using those associations, new semantically valid mappings can be generated. Hence, the mapping universe is expanded. However, they are not added to the set of existing mappings. The reason is that there is no indication that they describe any of the intended semantics of the mapping system. This can be explained by the fact that there is no correspondence covered by any of the new mappings that is not covered by any of those that already exist. On the other hand, since the structure and constraints used by the existing mappings are not affected, there is no reason for adapting any of them.

Example 5.5 *Consider the case in which the source schema of Figure 3.1 is modified so that each company has nested within its structure the set of laboratories that the company operates. This introduces some new mappings in the mapping universe, for example, a mapping that populates the target schema only with companies that have laboratories. Whether this mapping should be used is something that cannot be determined from the schemas, or from the existing mappings. On the other hand, mapping* m_2*, which populates the target with companies independently of whether they have labs or not, remains valid and consistent.* ∎

In many practical cases, a part of the schema is removed, either because the owner of the data source does not want to store that information, or because she may want to stop publishing it. The removal of an element forces all the mappings that involve that element to be adapted.

We consider first the removal of atomic type elements. When atomic element e is removed, each constraint F in which e is used is removed by following the procedure described in Section 5.2.1. If the atomic element e to be removed is used in a correspondence V, then every mapping m that is covering V has to be adapted. More specifically,

Algorithm 5: Atomic Element Deletion Mapping Adaptation Algorithm

Input: A source schema $\mathcal{S}$
A target schema $\mathcal{T}$
A set of mappings $\mathcal{M}$ from $\mathcal{S}$ to $\mathcal{T}$
An atomic element e
Output: A new set of mappings $\mathcal{M}'$

REMELEMENT($\mathcal{S}$,$\mathcal{T}$,$\mathcal{M}$,e)
(1) **while** exists constraint F that uses e
(2) remove F
(3) **foreach** m:(**foreach** $A^{\mathcal{S}}$ **exists** $A^{\mathcal{T}}$ **with** D) $\in \mathcal{M}$
(4) $D \leftarrow \{q \mid c$ is correspondence covered by $m \wedge c$ is not using $e \wedge$
(5) q is the **with** clause of c $\}$
(6) **if** D=$\emptyset$
(7) $M \leftarrow M$ - $\{m\}$
(8) **return** $\mathcal{M}'$

---

the equality in the **with** clause of the mapping that corresponds to V is removed from the mapping. If mapping m was covering only V, then the **with** clause of m becomes empty, thus m can be removed. If the atomic element e is used neither in a correspondence nor in a constraint, it can be removed from the schema without affecting any of the existing mappings. Algorithm 5 describes the steps followed to remove an atomic element. To remove an element that is not atomic, its whole structure is visited in a bottom up fashion starting from the leaves and removing one element at a time, following the procedure described in Algorithm 5. A complex type element can be removed if all its attributes (children) have been removed.

Example 5.6 *In the mapping system of Figure 5.2, removing element* `year` *from* `project` *will not affect any mappings since it is used neither in a constraint nor in a correspondence. On the other hand, removing element* `gid` *from* `grant` *will invalidate mapping* m_1*, which populates the target with projects and their grant information. After the removal of element* `gid`*, mapping* m_1 *will be modified and the condition that involves* `gid` *will be removed from its* **with** *clause.*

Assume now that the element `cname` *is deleted. This will make mapping* m_2 *of the mapping system in Figure 5.2 invalid, and the condition* o.`company.cname`=w.`company.cname` *will be removed from the* **with** *clause by the mapping adaptation algorithm. Notice that after this removal, the element* `company` *does not contribute anymore through the mapping to the population of the target schema. However, our mapping adaptation algorithm will not remove the* `company` *element from the* **foreach** *clause. The reason is the preservation of the initial semantics of the mapping. The* `leader` *element of the target schema is populated through mapping* m_2 *with the names of the owners of companies. If the* `company` *element is removed from the* **foreach** *clause of mapping* m_2 *after the removal of element* `cname`*, then the target schema will be populated with names that may be of any* `person`*, and not only* `company` *owners. However, as stated previously, one of the main principles of the mapping adaptation algorithm is to preserve as much as possible the initial semantics of the mappings by performing the minimum required changes during the adaptation process. As such, the algorithm maintains the join between* `company` *and* `person` *elements to preserve the fact that leaders of the companies in the target schema should be company owners.* ■

Another common operation in schema evolution is updating the type of an element e to a new type t. This case will not be considered separately since it is easy to show that it is equivalent to removing element e and then adding a new one of type t with the same name, e.

5.2.3 Schema Restructuring

The last two sections described how the schema can evolve by changing its constraints or by adding/removing some of its parts. This section describes how we can handle a third kind that a schema can change. This is by modifying its structure without removing or adding elements. There are three common operations of this kind of evolution that we consider: rename, move and copy. The first renames a schema element, and it is mainly

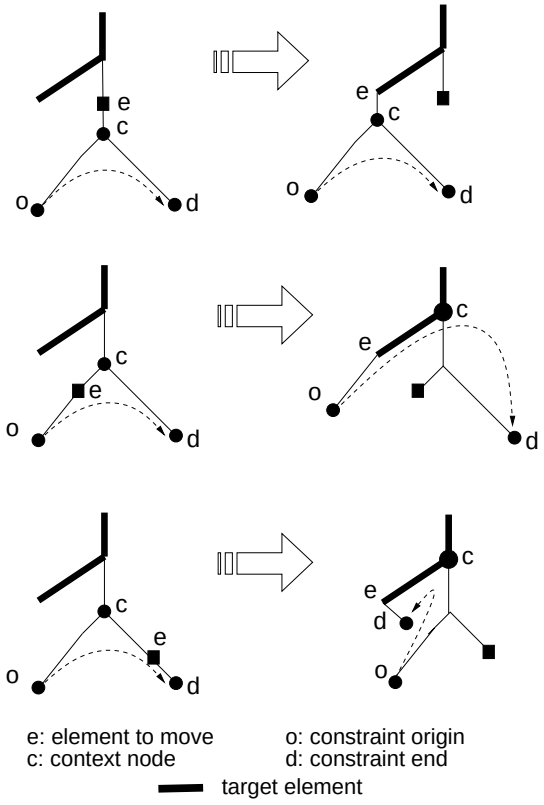


Figure 5.3: Updating a constraint after an element move

a syntactic change. It requires visiting all the mappings and updating every reference to the renamed element with its new name. The second operation moves a schema element to a different location, while the third does the same but moves a replica of the element instead of the element itself. When an element is copied or moved, it carries with it design choices and the semantics it had in its original location, i.e., schema constraints. Mapping selections and decisions that were used in the original location, should also apply in the new location.

Adapting Schema Constraints

Assume that a schema element e is to be moved to a new location. Due to this move, constraints that are using the element e may become invalid and must be adapted. Recall that the constraints we consider are NRIs that naturally include the constraints of

relational and XML schemas. A NRI constraint F is of the form **foreach** P_0 [**foreach** P_1 **exists** P_2 **with** C] where the P_1 and P_2 start from the last variable of the P_0 which represents the context element, and C is of the form e_1=e_2 where e_1 and e_2 are expressions depending on the last variable of P_1 and P_2, respectively. To realize how a constraint F is affected by the change, we have to consider the relative position in the schema of element e with respect to the context element of F in the schema structure. Figure 5.3 provides a graphical explanation of how F is adapted due to the move of element e. In the figure, for constraint F, we use c, o, and d to denote (both before and after the move) the context element, the element identified by the query select e_1 from P_0, P_1 (also called the origin element) and the element identified by the query select e_2 from P_0, P_2 (also called the destination element), respectively. If element e is an ancestor of the context node c, then the nodes c, o, and d move together with e. The modified constraint will have the form **foreach** P_0' [**foreach** P_1' **exists** P_2' **with** C] where P_0' is from the query determining the new context node c. The expression P_1' is the same as P_1 except that the starting expression is updated so that it corresponds to the new location of the context node (and a similar change applies to P_2' as well). If the context node is an ancestor of e, then e is either used in P_1 or in P_2. Assume that e is used in P_1. (The other case is symmetric.) This case is shown in the second part of Figure 5.3. The node o moves rigidly with e to a new location, while d remains in the same position. We then compute a new context node as the lowest common ancestor between the new location of o and d. The resulting constraint is then **foreach** P_0' [**foreach** P_1' **exists** P_2' **with** C'] where P_0' determines the new context node and P_1' and P_2' are relative to query P_0' and determine the new location of o and d. The C' is the result of changing C so that it uses the end points of paths P_1' and P_2'.

Example 5.7 *Assume that the designer of schema S in the mapping system of Figure 3.1 has decided to store grants nested within each company so that each company contains the set of its grants. This translates to a move of the element* `grants` *to a position under*

the element company. *Consider the constraint* f_2, *which specifies that each grant having a government sponsor refers to its contact information. Once the grants are moved, this constraint becomes inconsistent since there are no grant elements under the schema root* S. *To adapt the constraint, we use the previously described steps: we are in the second case shown in Figure 5.3, in which the element that moves is between the context element* c, *the root in this case, and* o. *The element query of element* grants *in its new location is:*

select w.company.grants from S.companies w

The variable binding of w *does not exist in* f_2 *so it is appended to the* from *clause of the above query, and every reference to expression* S.grants *is replaced by the expression* w.company.grants. *The final form of the adapted constraint* f_2 *is shown below. (The* **exists** *clause need not be changed, since the new context element continues to be the root.)*

```
foreach S.companies w, w.company.grants g,
        g.grant.sponsor→government r
exists  S.contacts c
with    c.contact.cid=r
```

■

Adapting Mappings

When an element is moved to a new location, some of the existing logical associations that were using the element become invalid and new ones have to be generated. To avoid redundant recomputation by regenerating every association, we exploit information given by existing mappings and computations that have already been performed. In particular, we first identify the mappings that need to be adapted by checking whether the element that is moved is used in either of the two associations on which the mapping is based. Let A be an association, which is using the element e that is about to move, and let t be

the element in its new location. More precisely, assume that e and t have the following forms:

$$e: \underline{\textsf{select}}\ e_{n+1}\ \underline{\textsf{from}}\ e_0\ x_0,\ e_1\ x_1,\ ...,\ e_n\ x_n$$
$$t: \underline{\textsf{select}}\ t_{m+1}\ \underline{\textsf{from}}\ t_0\ y_0,\ t_1\ y_1,\ ...,\ t_m\ y_m$$

We first identify and isolate the element e from association A, by finding the appropriate renaming from the **from** clause of e to A. For simplicity, assume that this renaming is the identity function, that is, A contains literally the **from** clause of e. In the next step, the **from** clause of t is inserted at the front of the **from** clause of A. We then find all usages of e_{n+1} within A, and replace them with t_{m+1}. After these replacements, it may be the case that some (or all) of the variables $x_0, \ldots, x_n$ have become redundant (i.e., not used) in the association. We eliminate all such redundant variables. Let us denote by A' the resulting association.

Since the element t in the new location may participate in its own relationships (based on constraints) with other elements, those elements have to be included as well in the new adapted version of association A'. We do this by chasing A' with the schema constraints. The chase may produce multiple associations $A'_1, \ldots, A'_k$ (due to choice types). Finally, any mapping using the old association A, say **foreach** A **exists** B **with** D, is removed from the list of mappings, and is replaced with a number of mappings m_i: **foreach** A'_i **exists** B **with** D' one for each association A'_i. The conditions D' correspond to the correspondences in D plus any additional correspondences that may be covered by the pair $<A'_i, B>$ (but not by the original pair $< A, B >$).

As an important consequence of our algorithm, all the joins that were used by the original mapping and were not affected by the schema change will be used, unchanged, by the new, adapted, mapping. Hence, we preserve any design choices that might have been made by a human user based on the original schemas. We illustrate the adaption algorithm with the following example.

Example 5.8 *Assume that, in the mapping system of Figure 5.2,* `grants` *are moved*

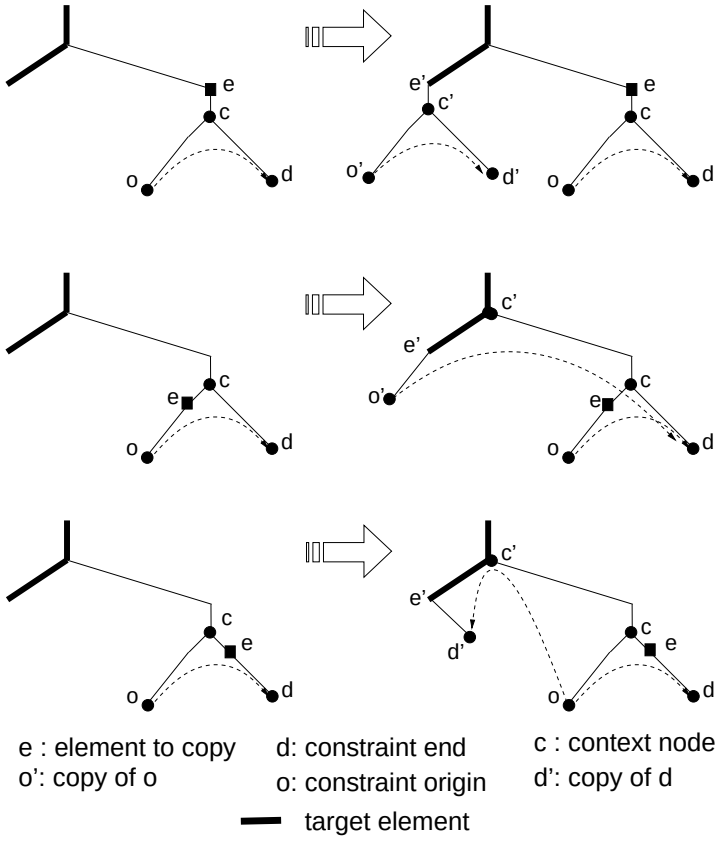


Figure 5.4: Introducing a new constraint after an element copy

under `company` *as in Example 5.7. This change affects neither mapping* m_3*, nor mapping* m_2*. (Recall from Section 5.2.2 that just the addition of new structure,* `grants` *in this case, for* m_2 *does not require* m_2 *to be adapted.) However, mapping* m_1*, based on the logical association* $A_3^{\mathcal{S}}$ *(see Figure 4.3), is affected. First, schema constraints are adapted as described in Example 5.7. Then we perform the mapping adaptation steps for*

e: select `S.grants` from *and*
t: select w`.company.grants` from `S.companies` w

(Note here that the from *clause is empty.) The clause "*`S.companies` w*" is added in the* from *clause of* $A_3^{\mathcal{S}}$*. Next, all occurrences of* `S.grants` *are replaced by* o`.company.grants`*. After this, the resulting association is chased with the source schema constraints. The (adapted) constraints* f_1, f_2*, and* f_3 *are already satisfied, and hence not applicable. However,* f_4 *and* f_5 *will be applied. The chase ensures coverage of the two correspondences*

on `cname` *and* `name`*, the last one in two different ways. Hence, two new mappings are generated. The first is:*

m'_{1a}: **foreach** S.companies w, S.projects p,
w.company.grants g, g.grant.sponsor→government r,
S.contacts c, S.persons s
where p.project.name=g.grant.project and
r=c.contact.cid and
w.company.CEO=s.person.SSN
exists T.govProjects j, j.fundings u,
T.finances f, T.companies o
where u.funding.finId=f.finances.finId and
f.finances.company=w.company.cname
with o.company.cname=w.company.cname and
o.company.leader=s.person.name and
f.finances.sponsorPhone=c.contact.phone and
f.finances.amount=g.grant.budget and
u.funding.fid=g.grant.gid and
u.funding.pi=g.grant.pi and
j.govProject.code=p.project.code

while the second is the one that considers the owner of the company as a leader instead of the CEO. Note how the algorithm preserved the choice made in mapping m_1 *to consider government projects, and how the initial relationships between* `projects` *and* `grants`*, as well as* `grants` *and* `contacts` *in mapping* m_1 *were also preserved in the new mapping.* ■

In the above analysis, we considered the case of moving an element from one place in the schema to another. In the case that the element is copied instead of being moved, the same reasoning is used and the same steps are executed. The only difference is that the original mappings and constraints are not removed from the mapping system as in the case of a move. Schema constraints and mapping choices that have been made continue to hold unaffected after a structure in the schema is copied. Figure 5.4 demonstrates how a new constraint is introduced when an element is copied.

5.3 A Ranking Mechanism for Rewritings

In the previous sections, we presented a methodology for adapting mappings that are affected by a change to the schema structure or constraints. This methodology detects the mappings that are affected by the change, and generates a number of semantically valid rewritings. All those rewritings are consistent with the semantics of the schemas, the structure, and also the user choices that have been made in the past and are encoded in the current mappings. An affected mapping can be replaced with one, two, or even all of its generated rewritings. The existence of more than one candidate is natural since the new modified schema has new semantics. However, there are cases in which not all of the generated mappings describe the semantics of the intended data transformation between the two schemas. Selecting only those that do requires human intervention. In order to assist the user in this selection, we developed a model for systematically ranking the rewritings. A number of approaches in the literature [CJR98, MD96] have used the cost of updating a materialized target as a measurement of the importance of the mapping. Although the maintenance cost is important, in our context, since our main goal is to preserve, as much as possible, the semantics of the mapping that is to be adapted, we believe that a different criterion is more appropriate. For that, we introduce a new model for measuring the similarity of two mappings.

The similarity measure is based on the observation that two mappings are semantically similar if they use common schema elements. The more schema elements and equalities they have in common, the more (semantically) similar they are. On the other hand, the more schema elements in which they differ, the less (semantically) similar they are.

Definition 5.9 *Let $|m|$ represent the sum of the number of schema elements used in a mapping m and the number of equalities between these elements in that mapping. The* relative similarity *of mapping m_1 to mapping m_2 (noted as $\mathcal{S}(m_1, m_2)$) is defined as the weighted sum:*

$$\mathcal{S}(m_1, m_2) = \rho_1 \frac{|m_1 \cap m_2|}{|m_2|} + \rho_2 \frac{|m_1 \cap m_2|}{|m_1|} \tag{5.1}$$

where $|m_1 \cap m_2|$ *is the number of elements and equalities that are used in both mappings* m_1 *and* m_2*. An equality is* used *in a mapping when it appears in its* where *or* **with** *clause, or it is implied by them. The quantities* ρ_1 *and* ρ_2 *are constant values for which* $\rho_1+\rho_2=1$*.*

Intuitively, the first fraction of the *relative similarity* $\mathcal{S}(m_1, m_2)$ measures how many elements mapping m_1 has in common with mapping m_2, while the second measures how many surplus elements mapping m_1 has compared to mapping m_2. Factors ρ_1 and ρ_2 determine the relative importance of the two fractions in the relative similarity.

In some cases, the relative similarity is not enough to determine if one mapping is preferable to another. Consider, for example, the case where a new constraint is added to one of the schemas. A mapping m_o needs to be rewritten, since one of its associations is affected by this change. The association is chased and a set of new associations results, each one of which is used to form a new candidate rewriting. To rank the rewritings, the relative similarity of each mapping and mapping m_o is calculated. From the way the rewritings were generated, they will all contain the elements and conditions of the mapping m_o. Thus, deciding which one is preferable depends on the number of surplus elements and conditions. For two or more mappings, that number may be the same, making their relative similarity to m_o be the same. In order to be able to select one over another, we introduce a new quantity called the support level. Intuitively, the support level utilizes knowledge of the other mappings to decide which of the rewritings that have the same relative similarity is most likely closest to the semantics that the user has in mind. It does so, by computing the relative similarities of the rewriting with every other mapping and selects the one with the highest relative similarity.

Definition 5.10 *Let* M *be a set of mappings. The* support level *of mapping* m *from the*

set M is defined as:

$$\mathcal{L}_M(m) = \frac{\Sigma_{m' \in M} \mathcal{S}(m, m')}{|M|} \tag{5.2}$$

Example 5.11 *Consider the two mappings m'_{1a} and m'_{1b} in Example 5.2, which were generated as a result of the addition of a new constraint. Recall that the difference between those two mappings is that the first one considers the CEO to be the leader of the company while the second considers the owner to be the leader. Both these mappings have a relative similarity 0.869 to mapping m_1. However, one can observe that the existing mapping m_2 considers the owner of the company to be the company leader. Based on that, the rewriting mapping m'_{1b} seems to be more appropriate than m'_{1a}. Indeed, the support level of mapping m'_{1b} is 0.634 which is higher than the 0.624 support level of mapping m'_{1a}.*

Note that the ranking mechanism is not used to eliminate any of the rewritings. All the mappings generated by the algorithms presented in the previous sections are semantically valid mappings and valid rewritings. The ranking mechanism is used to rank these mappings according to what the system judges to be the most likely (based on the schema structure, the semantics, and most of all, the existing mappings).

Unfortunately, it turns out that the relative similarity is not a metric.

Theorem 5.12 *The relative similarity is not a metric.*

Proof. Consider the case where $|m_1|$=6, $|m_2|$=10, $|m_3|$=10, $|m_1 \cap m_2|$=0, $|m_2 \cap m_3|$=5, $|m_1 \cap m_3|$=5, ρ_1=$\frac{1}{2}$ and ρ_2=$\frac{1}{2}$. Substituting the values in the formula, it can be seen that: $\mathcal{S}(m_1, m_2) + \mathcal{S}(m_2, m_3) = 0 + 0 + \frac{1}{2}\frac{1}{2} + \frac{1}{2}\frac{1}{2} = \frac{1}{2} < \frac{2}{3} = \frac{1}{2}\frac{5}{10} + \frac{1}{2}\frac{5}{6} = \mathcal{S}(m_1, m_3)$ ∎

We have to note here that the ranking is not used to decide whether a mapping should be eliminated or not, but only to show the rewritings in some order. If a metric measure is required, one could use only one of the two terms in the definition of the relative similarity. Alternatively, one could use some other similariy measures. It is among our future research plans to study them and find those that are more appropriate for our case.

5.4 Analysis

The previous sections described the various schema changes we consider and how we can handle them. In this section, we analyze and identify properties of the algorithms we described.

A main performance feature of the mapping adaptation approach we presented is the incremental maintenance of the mappings, which leads to fewer computations than would be required if the mappings were regenerated from scratch after the schemas are modified. For the regeneration, a mapping generation tool, like Clio, could have been used. First, a user would have to enter the correspondences between the two schemas again. Then, the mapping generation tool, e.g., Clio, would have to generate all the mappings in the mapping universe, and the expert user would have to select only those that describe the intended semantics of the transformation. It is our hypothesis that a great deal of these semantics are embodied in the existing mappings, but this knowledge is lost once the old mappings are dropped. Our mapping adaptation approach uses this knowledge to avoid regeneration of mappings that would have been rejected anyway by the user, and also to avoid computations that will give results already available in the existing mappings. For example, when a new constraint is added, and an association of a mapping needs to be updated, the chase does not start from the structural associations, as a mapping tool like Clio [PVM+02b] would have done, but from the existing associations in the mapping.

To determine what mappings are affected by a schema change the mapping adaptation algorithm checks for dominance. Dominance is checked by looking for one-to-one renaming functions between the variables of two queries. Finding such a renaming in the general case requires time exponential in the size of the queries. However, the NRI constraints and the schema element query representation are based on linear path queries for which a renaming one-to-one function can be found (if it exists) in linear time. Furthermore, the problem of finding whether an association A is dominated by an association B, when either A or B has an empty where clause, can be reduced to the problem of

finding subtree isomorphism which requires time $O((k^{1.5}/\log k)n)$ [ST99], where n is the number of nodes of the tree representation of the schema and k is a constant.

Theorem 5.13 *The rewritings generated by the mapping adaptation algorithm are semantically complete mappings.*

Proof. As we presented in Section 5.2, the generated rewritings are constructed by computing a logical association for the new version of the modified schema and then combining it with the covered correspondences. Hence, by construction, the mappings generated by this adaptation process belong to the new mapping universe $\mathcal{U}^{\mathcal{C}^n}_{\mathcal{S}^n,\mathcal{T}^n}$. ∎

What is not clear is what mappings among those in the new mapping universe are chosen as rewritings of a mapping affected by a schema change. To understand this, consider schemas and mappings as graphs. A schema can be modeled as a tree, as we saw in Section 3.2. Every structural association can be modeled as a subtree of that graph, with the same root. A logical association or a mapping can then be modeled as a connected graph that consists of a set of structural association graphs with some edges added between their nodes that make them connected. Each such edge represents an equality in the where clause (if the two nodes it connects belong to the same schema), or an equality in the **with** clause. Schema change operators can then be modeled as graph operations.

Theorem 5.14 *The rewriting generated as a result of an addition or removal of a new schema element is the one whose graph has the minimum tree-edit distance to the graph of the original mapping.*

Proof. Addition of a new schema element under an element e translates to the insertion of a new node connected to the node that models element e. This means that the tree-edit distance between the original mapping and the rewriting is one, which is the minimum. Deletion of an element translates to graph pruning, hence, it is also one, i.e., minimum. ∎

Theorem 5.15 *The rewriting after the addition of a new constraint results to a new mapping whose graph representation contains the original mapping graph and has the minimum tree-edit distance from it. After the removal of a new constraint, the graph consisting from the graph representation of the two rewritings is the one that has the minimum edit distance from the graph representation of the original mapping, compared to any other graph consisting of two graph representations of mappings in the new mapping universe.*

Proof. A constraint insertion is modeled as an introduction of a new edge in the graph. Let G_o, G_n be the graph representations of the original mapping and its rewriting after the insertion of the new constraint. Let $G_e = G_n - G_o$ be the part of G_n that is not in G_o. This part was added as a result of the chase. Hence, every mapping in the new mapping universe that contains G_o will also contain G_e. Among all these mappings, the one the mapping adaptation algorithm chooses is the one that has nothing else apart from G_o and G_e, which means that the tree-edit distance is zero i.e., minimum.

A constraint removal is modeled as the removal of an edge from the graph representation of the mapping. This removal leaves the graph disconnected. From the set J of all the associations of the new schema that are subsumed by the association that became invalid, the algorithm selects those that are not subsumed by any other association in that set J. This translates to the two maximal connected components of the original mapping graph after the removal of the edge. This means that the graph representations of the two rewritings together will have a tree-edit distance equal to one from the original mapping graph representation, i.e., minimum. ∎

5.5 A Mapping Adaptation Tool

To evaluate the effectiveness and usefulness of our approach, we have implemented a prototype tool called ToMAS, which stands for **To**ronto **M**apping **A**daptation **S**ystem.

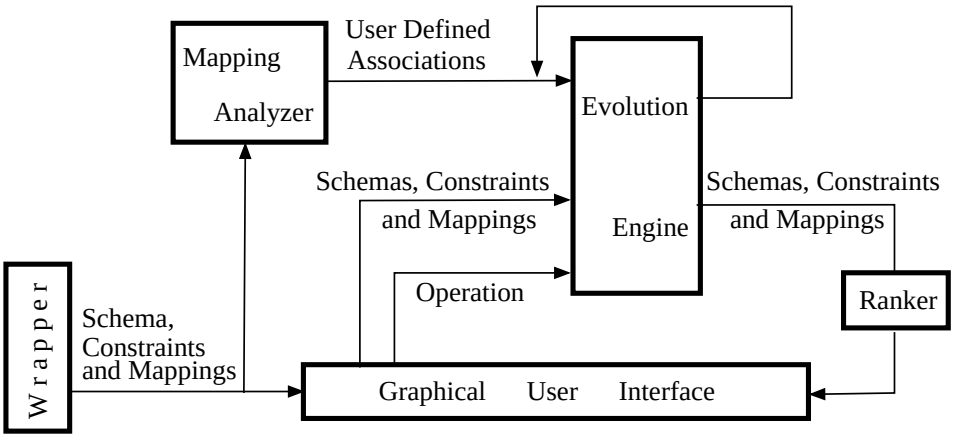


Figure 5.5: ToMAS architecture

ToMAS has been developed at the University of Toronto. We have applied ToMAS in a variety of real application scenarios. In this section, we describe ToMAS architecture and we report our experience using it to adapt mappings between real schemas. The results of the experiments indicate that indeed it is worth using a mapping adaptation system like ToMAS to automatically maintain the mappings between schemas. Specifically, we show that (i) the time needed for incrementally updating the mappings under schema changes is negligible, and (ii) this incremental adaptation requires much less effort than a "from-scratch" rebuilding of the mappings. At the end, we highlight the benefits of using a mapping adaptation tool in concert with a physical design tool that manipulates schemas. In particular, we show how our tool can facilitate the selection of a good relational schema for storing XML data. We need to note here that the experiments were not conducted by real users but by us. The semantics of the sample schemas we used were such that our knowledge and experience were adequate to allow us to judge whether the rewritings generated by ToMAS were correct. Furthermore, our knowledge of the algorithm details allowed us to evaluate better the behavior of the system, and understand its decisions in cases where the results were not what we initially had thought.

5.5.1 Architecture

ToMAS follows a modular architecture as can be seen in Figure 5.5. At the heart of ToMAS is the *Evolution Engine* that contains implementations for each of the evolution operators we described earlier. ToMAS manages a mapping system (a pair of schemas and a set of mappings between them). As the schemas evolve, ToMAS detects mappings that may need to be updated and, based on the schema constraints and the user associations, it creates a set of potential rewritings that are consistent with the modified schemas.

To update the mappings, apart from the schema and constraint information, the evolution engine has to know the user associations that are contained in the mappings. This information is provided by the *Mapping Analyzer*. The Mapping Analyzer constructs all the structural associations of the schemas and chases them to generate the set $\mathcal{A}_{\mathcal{S}}$ of logical associations that are based on the schema structure and constraints. Each association A used in an existing mapping is then chased to become a logical association and make the mapping a semantically valid mapping. If the resulting logical association is not in the set $\mathcal{A}_{\mathcal{S}}$, it means that the association A that the mapping was using represents a choice that has been explicitly stated by the user, thus, it is a *user association*. The set of user associations that are found, along with the semantically valid mappings, is provided to the Evolution Engine. This step is performed once at the beginning of the evolution process, right after the schemas are loaded.

The system is equipped with a set of pluggable schema *wrappers* used to import schemas and mappings from different data models into the internal nested relational representation. Currently, relational and XML wrappers have been implemented.

The schemas and the mappings are presented to the user through a *Graphical User Interface* (Figure 5.6). Through this interface, the user may also modify the schemas. For each modification, the schemas, the requested modification and the existing mappings are provided to the evolution engine, which updates both the mappings and the schemas. The results are returned back to the interface for presentation to the user and further

modifications.

The last component of ToMAS is the mapping *Ranker*. Its role is to rank the candidate rewritings generated by the evolution engine according to the ranking criterion that was presented in Section 5.3, before sending them to the Interface.

In its current implementation, ToMAS assumes that the schemas have no cyclic constraints and no recursive types. If the schemas have them, ToMAS will enter an infinite loop state. To avoid this, a new module can be added that checks for cyclic constraints and recursive types when the schemas are loaded.

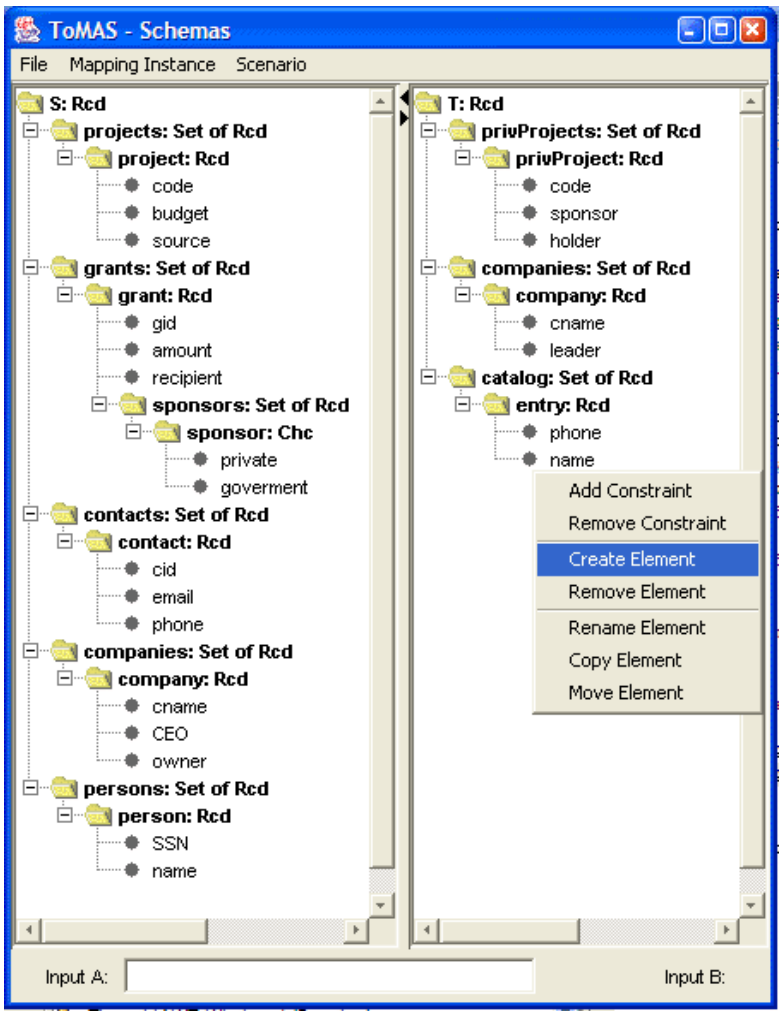


Figure 5.6: ToMAS user interface

Schema	Size	Corresp/ces	Mappings
ProjectGrants	16 [6]	6	7
DBLP	88 [0]	6	12
TPC-H	51 [10]	10	9
Mondial	159 [15]	15	60
GeneX	88 [9]	33	2

Table 5.1: Test schemas characteristics

5.5.2 Performance

We now investigate the efficiency of our proposed incremental mapping adaptation algorithms. We conducted a series of experiments on some schemas, both relational and XML, that vary in terms of size and complexity. Their characteristics are summarized in Table 5.1. The size is shown in terms of atomic schema elements, and within the brackets we give the number of schema constraints. We used two versions of each schema, and generated mappings from the first version to the second. In some cases, the two different versions of a schema were available on the Web (representing two different evolutions of the same original schema). In other cases, where a second version was not available, it was manually created. Using the Clio mapping generation tool, a number of correspondences were used to generate a set of semantically meaningful mappings between each pair of schemas (the last two columns of Table 5.1 indicate their exact numbers). From them, two mappings were selected as those representing the intended semantics of the correspondences. These mappings were the mappings to be maintained through schema changes.

An arbitrary sequence of changes on randomly selected schema elements was generated and applied to each schema. Even for only two mappings, due to the large size of the schemas, it was difficult for a user to understand how the mappings were affected by the changes, and how they should be adapted. We considered two alternative adaptation techniques. The first was to perform all the necessary modifications on the schemas and at the end use a mapping generation tool (specifically, Clio) to regenerate the mappings. Due to the fact that the names of some attributes might have changed and elements

might have moved to different places in the schema, schema matching tools were unable to re-infer the correspondences. This means that the correspondences had to be entered manually by the user. Once this was done, the mapping generation tool produced the complete set of semantically valid mappings, and the user had to browse through all of them to find those that described the intended semantics. The second alternative was to perform the schema changes and let ToMAS maintain the mappings. For the experiments we performed, we chose to maintain 10% of the generated mappings. This means that we assumed that during the mapping generation process, the data administrator had browsed through the mappings generated by the mapping generation tool (the numbers are indicated in the last column of Table 5.1) and chose 10% of them as these that represent the intended semantics of the correspondences. ToMAS was then used to maintain these two mappings while schemas were evolving. ToMAS returned only a small number of mapping rewrittings, since it utilized knowledge about choices that were embedded in the initial set of mappings. At the end, the user had to go through only the small number of adapted mappings (that were ranked according to our ranking criterion) and verify their correctness. We performed and compared both techniques experimentally. It is in our future research plans to provide a more complete study on how the results we present in the following two subsections are affected by modifying the number of mappings that are considered (the 10% percentage) and the number of correspondences.

Time Performance

For each change, we measured the time needed for the schema to be updated and the mappings to be adapted. Figure 5.7 summarizes the results of this experiment. The time indicated for each operator is the average time of this kind of operator in the mapping system. What we noticed was that the time of each operator depends not only on the nature of the schemas and the mappings but also on the sequence of changes that have taken place before. The reason is that that operators have different tasks to perform

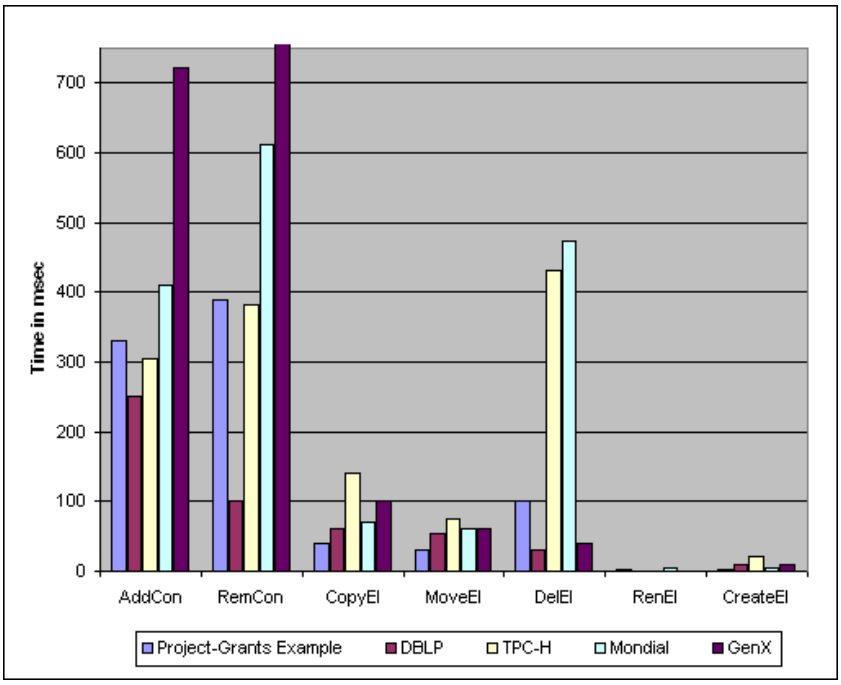


Figure 5.7: Average mapping adaptation times

depending on the schema to which they are applied. We calculated the average time of completion for each operator over all the times it was performed and we present it in Figure 5.7.

Creation of new elements in the schema does not require any updates on the mappings. However, the inserted structure has to be type-checked for consistency before being inserted in the schema. This checking causes the create element operation to take slightly more time compared to renaming.

Copying and moving an element are mostly syntactic modifications. They require deletion of the elements to be moved (copied) in a mapping or in a constraint. At the end of the move (copy), some chasing is required at the new location. This chasing accounts for the time difference between these two restructuring operators and the element renaming or creation operators. TPC-H is the schema with the largest number of constraints (relative to the size of the schema) which makes chasing take longer than in any

other schema. On the other hand, if we exclude the ProjectGrant schema, which is quite small, the shortest adaptation time is achieved in DBLP, which is the only schema with no constraints.

The addition and removal of constraints, as expected, are expensive operations since they involve chasing. Adding a new constraint requires chasing an affected association, while removal involves the reconstruction of a number of structural associations, their chase, and the checking of whether the results of the chase are subsumed by the affected association. All those operations make mapping adaptation under constraint removal more expensive than adaptation under constraint addition. Surprisingly, in every test we have conducted on the DBLP schema, the constraint removal take less time than the constraint addition. This may be because DBLP is the only schema that initially has no constraints. Constraints that are removed are those that have been generated through an "addConstraint" operation, and thus do not create long chains of consecutive constraints. As a consequence, the chase required at most one chase step. Furthermore, DBLP is very shallow, so the time required for the construction of the structural associations was almost negligible. An interesting observation is that only the GeneX schema under constraint removal took more than one second, even though there was only one mapping that had to be updated. The reason is that the mapping was really large, involving 29 joins, hence, the corresponding association had 29 variables. Finding one-to-one renamings was the expensive operation in this complex mapping.

Deletion of an element requires first the removal of the correspondences and the constraints that involve the element. This means that the cost of an element deletion is at least as much as the cost of constraint removal. On the other hand, if the element is not used by any constraint, it can be removed in time that is almost equal to the time of renaming. On average, the high cost of the first kind of removal operation is balanced by the low cost of the second kind, so that the average performance is as shown in Figure 5.7.

The above analysis shows that, despite the size and complexity of the schemas, the

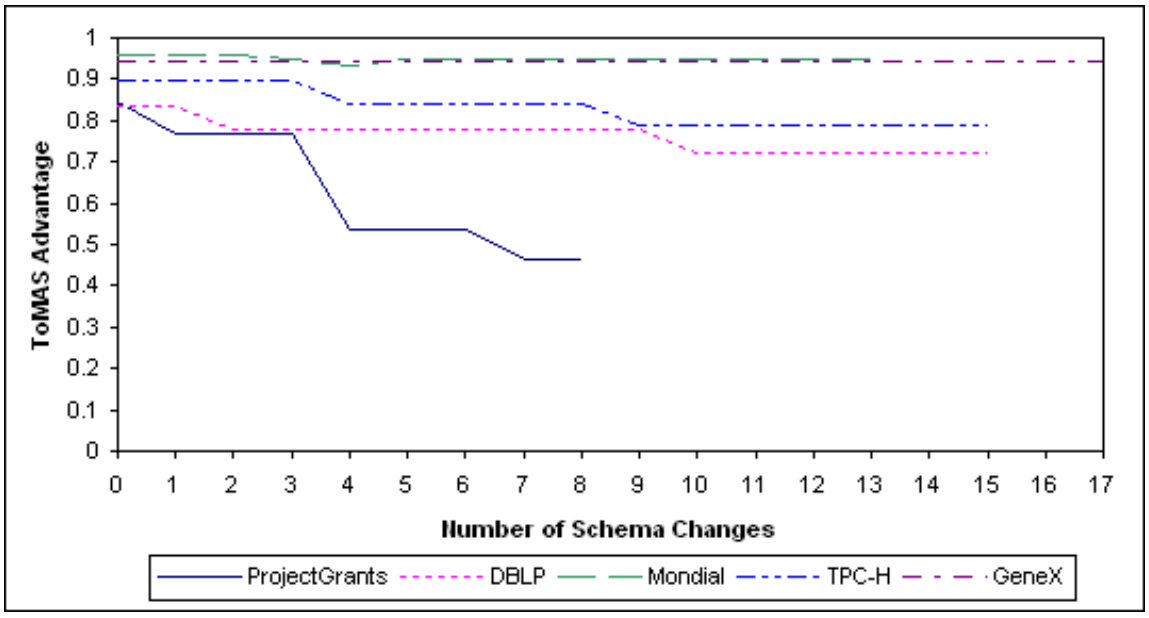


Figure 5.8: The benefit of ToMAS for different schemas as a function of the number of changes

time for mapping adaptation is short enough to permit its use in interactive applications.

Benefit.

We compared the user effort required in the two approaches. In the first approach, where mappings have to be regenerated from scratch, the effort of the user was measured as the number of correspondences that have to be re-specified, plus the number of mappings that the mapping generation tool produces and which the user has to browse to select those that describe the intended semantics of the correspondences. If ToMAS is used, however, the effort required is just the browsing and verification of the adapted mappings. As a comparison measurement, we used the following quantity, which quantifies the advantage of ToMAS against the "from-scratch" approach.

$$1 - \frac{mappings\ generated\ by\ ToMAS}{mappings\ generated\ by\ the\ mapping\ tool\ +\ correspondences}$$

A value of 0.7, for example, means that ToMAS saves 70% of the effort required in the "from-scratch" alternative. Figure 5.8 provides a graphical representation of how the ToMAS advantage changes as a function of the number of changes for the various schemas. As the number of changes becomes larger, and the modified schemas become very different from their original versions, the advantage of ToMAS is reduced, but the rate of reduction is small. Notice, for example, the line that corresponds to the ProjectGrants example used in this work. After 8 changes, ToMAS has lost almost half of its advantage. The reason is that the schema is small, and after 8 changes, it has been completely restructured, describing new data semantics. In such a case, it makes no sense to try to preserve the transformation described by the initial mappings (as ToMAS does) while the schemas continue to evolve. For the larger Mondial and GeneX schemas, even after 17 changes, ToMAS still has an advantage. The reason is that, even after the 17 changes, the schema has not changed that much and still describes the semantics of the data as they were initially, so it makes sense to try to preserve the initial transformation. Fortunately, practice has shown that schemas do not change radically. The new evolved schemas are not dramatically different from their original version, and in these cases, ToMAS helpful in reducing the human effort. Of course, there is no formally defined notion of a "dramatic change", neither of a "large" schema. It always depend on how the schemas are used. We have noticed that there are schemas, like the GeneX, for which data administrators find a hard time defining mappings, and maintaining the mappings when the schemas change. For such schemas and these changes the data administrators are in a need of using tools to automate the generation and maintenance process, and then spend some extra time veryfing the results that were automatically generated. We consider such schemas "large" and the changes "not dramatic".

It is among our future plans to further investigate this issue in order to identify and have a clear understanding of the cases in which ToMAS is preferable to the Rondo approach, and vice-versa.

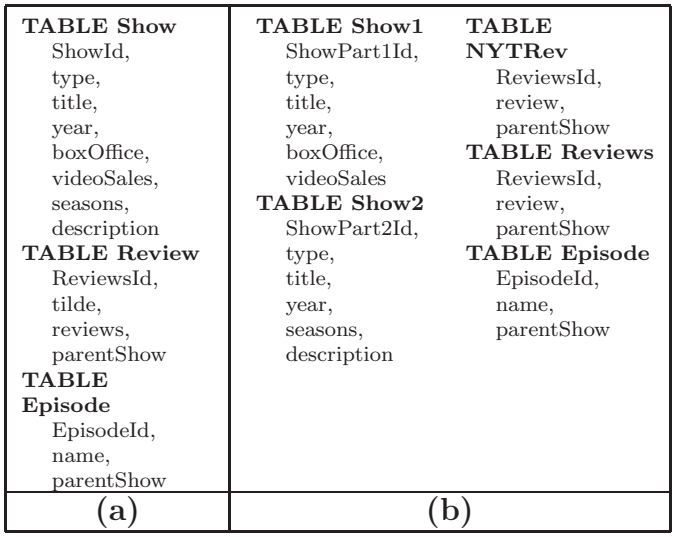


Figure 5.9: Two relational designs for the IMDB DTD

5.5.3 Case Study: ToMAS in Physical Design

We present our experience using ToMAS within one important application: physical data design. In the last few years, there has been a growing interest in storing XML data in relational database systems in order to be able to re-use the well-developed features (e.g., concurrency control, query processing, etc.) of such systems. A number of approaches have been proposed to resolve the mismatch between the nested semi-structured nature of the XML data and the relational model [FK99b]. Unfortunately, no approach is universally accepted, since none has been found to perform well in all the cases. LegoDB [BFH+02] is a physical database design tool for designing (optimized) relational storage structures for XML data. LegoDB helps a user to evaluate some of the many XML-to-relational "shredding" options [FK99b]. Since different XML-to-relational translations are best for different workloads and data characteristics, LegoDB provides an automated wizard for finding good relational designs. For this case study, we use an Internet Movie Database DTD example [BFH+02].

```
<!ELEMENT imdb (show*, director*, actor*)>
<!ELEMENT show (title, year, reviews*,
      ((boxOffice, videoSales) |
```

(seasons, description, episode*)))>

The two relational schemas of Figure 5.9 represent the results of two different shredding methods for the above DTD. Mappings between each of these two schemas and the DTD might be output by a design tool or might be created with a mapping tool like Clio. Although this is a very simple example, it is important to note that the mappings from the schema in Figure 5.9(a) to the DTD are quite complicated. Figure 5.10 shows the mapping in XQuery form, and indicates the complexity of the mapping, which has eight joins among the three participating tables of the source schema and a nesting depth two. So, even for this simple example, generating the correct mapping is clearly hard if it were to be done manually.

In tools like LegoDB, each shredding method requires a corresponding mapping. For the second way of shredding of Figure 5.9(b), for example, the mapping to the DTD has to be generated in advance and hard-wired in the LegoDB cost engine. Manual generation of such mappings, or generation through a mapping tool can be avoided with the use of ToMAS. Using ToMAS, one can take the DTD, the first shredding method, and the generated mapping of Figure 5.10 and start modifying the schema to bring it into the different form of Figure 5.9(b). Throughout this process, ToMAS maintains the mappings, and at the end, it will be able to provide automatically the mapping from the relational tables for any of the new shredding methods on the DTD.

We have performed the above test. After most operations, the mappings could be automatically updated without user intervention, but for some operations, there was a choice of what semantics to use. During the entire evolution, a ToMAS user had to make very few choices, and we were able to verify that the resulting mapping is the one we would have created if the adaptation had been done, or by using a mapping tool like Clio.

Our approach permits design tools like LegoDB to explore new storage schemes that might not be part of their (predefined) search space of designs for efficiency reasons. For

example, using ToMAS, we can permit a designer to suggest a different, *ad hoc*, vertical or horizontal decomposition of the relations (one not suggested by the workload). If the cost-based engine of LegoDB selects such a user-provided design, ToMAS can generate the mapping needed to use the original XML Schema as a view over this design.

Additionally, our ability to transform the relational schema of Figure 5.9(a) to the one of Figure 5.9(b) indicates the kind of transformations we are supporting. We do not consider only simple structural changes that take place locally on a table or on a class. We can support complex schema modifications involving many schema structures (in the specific example, relations) like copying an attribute from one table to another. In short, we have found that, with our small set of primitive operators, we can support the majority of compound schema changes that exists in the literature [Ler00].

```
<imdb>
    FOR $x0 IN $doc/DB/Review, $x1 IN $doc/DB/Show
    WHERE $x0/ReviewsId/text() = $x1/ShowId/text()
    RETURN
        <Show>
            <type> $x1/type/text() </type>
            <title> $x1/title/text() </title>
            <year> $x1/year/text() </year>
            FOR $x0L1 IN $doc/DB/Review, $x1L1 IN $doc/DB/Show
            WHERE
                $x0L1/ReviewsId/text() = $x1L1/ShowId/text() AND
                $x1/videoSales/text() = $x1L1/videoSales/text() AND
                $x1/boxOffice/text() = $x1L1/boxOffice/text() AND
                $x1/type/text() = $x1L1/type/text() AND
                $x1/title/text() = $x1L1/title/text() AND
                $x1/year/text() = $x1L1/year/text()
            RETURN
                <review> $x0L1/reviews/text() </review>
            <boxOffice> $x1/boxOffice/text() </boxOffice>
            <videoSales> $x1/videoSales/text() </videoSales>
        </Show>
    FOR $x0 IN $doc/DB/Episode, $x1 IN $doc/DB/Show, $x2 IN $doc/DB/Review
    WHERE $x0/EpisodeId/text() = $x1/ShowId/text() AND $x2/ReviewsId/text() = $x1/ShowId/text()
    RETURN
        <Show>
            <type> $x1/type/text() </type>
            <title> $x1/title/text() </title>
            <year> $x1/year/text() </year>
            FOR $x0L1 IN $doc/DB/Episode, $x1L1 IN $doc/DB/Show, $x2L1 IN $doc/DB/Review
            WHERE
                $x0L1/EpisodeId/text() = $x1L1/ShowId/text() AND
                $x2L1/ReviewsId/text() = $x1L1/ShowId/text() AND
                $x1/seasons/text() = $x1L1/seasons/text() AND
                $x1/description/text() = $x1L1/description/text() AND
                $x1/type/text() = $x1L1/type/text() AND
                $x1/title/text() = $x1L1/title/text() AND
                $x1/year/text() = $x1L1/year/text()
            RETURN
                <review> $x2L1/reviews/text() </review>
            <seasons> $x1/seasons/text() </seasons>
            <description> $x1/description/text() </description>
            FOR $x0L1 IN $doc/DB/Episode, $x1L1 IN $doc/DB/Show, $x2L1 IN $doc/DB/Review
            WHERE
                $x0L1/EpisodeId/text() = $x1L1/ShowId/text() AND
                $x2L1/ReviewsId/text() = $x1L1/ShowId/text() AND
                $x1/seasons/text() = $x1L1/seasons/text() AND
                $x1/description/text() = $x1L1/description/text() AND
                $x1/type/text() = $x1L1/type/text() AND
                $x1/title/text() = $x1L1/title/text() AND
                $x1/year/text() = $x1L1/year/text()
            RETURN
                <episode> $x0L1/name/text() </episode>
        </Show>
</imdb>
```

Figure 5.10: An initial mapping for the IMDB

Chapter 6

Representing and Querying Data Transformations

Modern information systems often store data that has been transformed and integrated from a variety of sources. A common schema is used to facilitate querying. However, integrated and transparent access to the data may obscure the original source and semantics of data items. For many tasks, including data cleaning, it is important to be able to determine not only where data items originated, but also why they appear in the integration as they do and through what transformation they were derived. The problem of determining the origin of data, known as data provenance, has been studied at the data level. Given a tuple in the integration (or view), find the data in the source that provide evidence for the tuple. Furthermore, the increase in the number and complexity of the mappings makes the managing of mappings impractical and calls for higher level of abstraction for this task.

In this chapter, we consider data provenance at the schema and mapping level. We consider how we can declaratively describe the origin of data and the data transformations. In particular, we consider how to answer questions such as "what schema elements in the source(s) contributed to this value?", "through what transformations or mappings

was this value derived?" or "what data elements were derived through transformations sharing a certain property?". Towards this end, we elevate schemas and mappings to first-class citizens that are stored in a repository along with the data and can be queried. We do that by extending the nested relational data model we described in Chapter 3. After specifying the problem in Section 6.1, we describe our framework in Section 6.2. Data annotations are used in order to associate the data values with the meta-data information, and an extended query language, called MXQL, is developed that allows schemas and mappings to be queried as regular data. Section 6.3 investigates the properties of the proposed query language, and shows that by doing this, a new kind of lineage is introduced that uses schema elements instead of data values. Section 6.4 shows how the language can be implemented using existing technology. This way, we are able to manage not only the origin of the data, but also the transformations through which the data has been derived. In a case study, we describe our experience with the design and implementation of the proposed framework in a real life scenario where a repository is constructed by integrating data from a number of real estate data sources (Section 6.5).

The material of this chapter has been published in ICDE 2005 [VMM05].

6.1 Introduction: Querying Mappings

Modern information systems and e-commerce applications use data integration tools to locate, collect, translate and integrate information in order to provide the user with a unified view of the data. This integration is achieved through mappings. Data obtained from well-known sources such as standard agencies, government organizations, research institutes or private companies may be well-understood and accepted. However, data coming from the integration of independent, physically distributed, heterogeneous sources that may have been developed with different requirements in mind, are not always well-understood. Some of the original data semantics may be lost during the integration, and

new data may also be added in the process [BA00]. Furthermore, as a consequence of the transparent access provided by the integration, the notion of distinct sources and their structures often disappears from queries and results. Hence, searching the integrated information for data that is not only relevant, but also best suited to a task at hand, is a challenge.

On the other hand, everyday life includes numerous applications where users need to know and reason about the origin of data [BKT01, CW00, HQGW93, MRT98, WM90]. One reason may be to evaluate the quality of the retrieved results. Knowing from where each particular data element was drawn and how it was computed allows users to apply their own judgment to the credibility of that information and decide whether some particular data is a semantically correct answer to their query. In systems where information from multiple sources is used, such knowledge may assist in interpreting the data semantics and resolving potential conflicts among data retrieved from different sources. In several other emerging applications, the ability to analyze "what-if" scenarios in order to reason about the impact of the data coming from specific sources (or parts of them) is of paramount importance. For data that may have been generated by integration of multiple sources, one should be able to take different views of the same data based on how and from where it has been been generated.

The problem of determining the origin of a specific data element in an integrated view is known to the research community as the problem of *data lineage* [CW03] or *data provenance* [BKT01]. Most of the work on lineage tracing has concentrated on instance-level tracing, that is, finding the specific values in the base tables that justify the appearance of a data element in a view. One way the problem is approached is by developing methods to generate the right queries on the source schema to return the data elements that are "responsible" for the appearance of a data value in the view. To generate such a query, the information of the view instance and the view query is required. The challenging computation of the inverse of a view query [BKT02] may be

required. In many cases, such a costly computation may be avoided if the interest is not in the originating data values themselves, but in the schema elements used or the way in which the data was transformed. This new paradigm suggests a new kind of lineage that is at the schema level and uses schema elements and mappings instead of specific data values. Such information can be important to provide users with a better understanding of the semantics of the data, and can help in resolving semantic conflicts that may arise from the integration of data from disparate heterogeneous sources [BGF+97]. In large scale integration systems, where the integrated instance is populated from many large schemas through numerous complex mappings, transformation queries or views, schema level provenance is a required first step to identify the mappings that generated a specific data value in the integration and the schema elements to which it was applied. This information can be provided to the user, or can be used to compute the data-level provenance if the exact originating values need to be identified or can be used to locate a piece of data according to the mappings applied to them.

To compute schema-level lineage, schemas and mappings need to be stored, managed and allowed to be queried as regular data. Meta-data information has been used in many applications. Commercial relational database systems, for instance, store schema information and view definitions in specialized tables called system catalogs. In our work, we are exploring how to make this information available to a designer in a systematic way by providing facilities for querying not only catalog information, but also very general mappings used to integrate and transform data. We also permit the origin of data to be recorded. For example, if data is integrated using a view that is generated through a union of multiple queries, we ensure that provenance information is not lost when the data is merged, a capability that is typically not automatically provided by commercial materialized view facilities. Currently, in order to deal with this limitation, data administrators usually store, with the integrated data, meta-data information in an ad-hoc way by tagging or annotating specific parts of the data. Our goal here is to make

this process systematic by providing data annotations that have a well-known meaning and are guaranteed to be reliable. In that way, we offer a more comprehensive set of meta-data and the functionality of formulating queries that use meta-data information along with the actual data, in a transparent and consistent way. The need for this kind of expressiveness in query languages has already been recognized in the research community [LSS96, AS03], and this functionality is already making its first steps into commercial products [CGGL04]. Previous work has considered only schema information as meta-data, but in many applications, mapping level information is also important.

The goal of the work in this chapter is to go further and provide an integrated solution and a systematic way for uniformly representing and declaratively querying data, its schema-level provenance, and the mappings through which it has been derived. We address issues such as determining what schema components and data sources contributed to the generation of a specific piece of information, through what views or queries this data was generated, and what transformation these queries performed. To achieve this, schemas and mappings are elevated to first-class citizens in the repository and the query language. An important component of our work is the representation of not just the identity of mappings, but their internal, declarative structure to help designers understand, compare and debug mappings.

There are numerous application scenarios and environments that motivate our work and highlight the need for such a functionality.

- **Scientific Databases:** Scientific data available on the Web [BA00] may be retrieved by third parties, transformed and stored in local databases. For these databases, it is essential for anyone interested in the accuracy and timeliness of the data to know the original location from which a given piece of data was drawn, and the processing that has been applied to it.

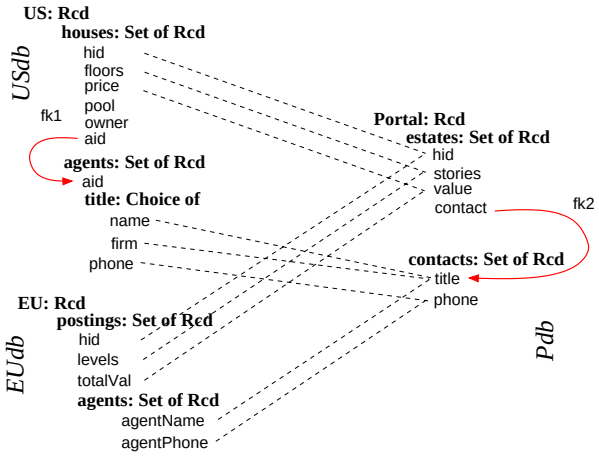
- **Information Portals:** A number of organizations and individuals are using in-

formation portals to discover, collect and organize information for presentation to the user in a consistent way. Knowing the sources and the transformations through which the portal data was derived is important in order to assess its quality [MRT98] since some of the originating sources may be known to contain inaccurate or out-of-date information, and some transformations may not accurately reflect the expected portal semantics.

- **Data Warehouses:** In data warehouses [CW00], views over source data are defined, computed, and stored in the warehouse to answer queries about the source data in an integrated and efficient way. Online analytical processing and data mining operations are applied to the data to perform analysis and predictions. In such applications, what-if scenarios play an important role. To support what-if scenarios, queries may be parameterized with conditions on originating schemas and view queries that control how the warehouse instance has been generated.

- **Peer-to-Peer Systems:** In peer-to-peer systems [TIM+03, KAM03], data passes through many peers and is transformed in various ways before reaching the peer that requested it, making it easy for an answer set to be contaminated with inaccurate or dirty data. Information on how, why and from where each data element appeared in the answer set may help detect and eliminate possible dirty data.

To better understand the problem, consider the case of a specific portal that integrates information from various real estate sites from around the world. Figure 6.1 indicates a portion of the schema of a European (`EUdb`) and an American (`USdb`) data source, as well as a portion of the portal (`Pdb`) integrated schema. The dotted arrows between the two schemas indicate how their elements correspond. The portal is populated with data retrieved from the two sources through the three mappings m_1, m_2 and m_3 indicated in Figure 6.1. The first mapping populates the integrated schema with data from the American site, in particular, house entries and independent agents. The second performs



m_1: **foreach** US.houses h, US.agents a, a.title→name n
where h.aid=a.aid
exists Portal.estates e, Portal.contacts c
where e.contact=c.title
with e.hid=h.hid and e.stories=h.floors and
e.value=h.price and c.title=n and
c.phone=a.agent.phone

m_2: **foreach** US.houses h, US.agents a, a.title→firm f
where h.aid=a.aid
exists Portal.estates e, Portal.contacts c
where e.contact=c.title
with e.hid=h.hid and e.stories=h.floors and
e.value=h.price and c.title=f and
c.phone=a.phone

m_3: **foreach** EU.postings p, p.posting.agents a
exists Portal.estates e, Portal.contacts c,
where e.contact=c.title
with e.hid=p.hid and e.stories=p.levels and
e.value=p.totalVal and c.title=a.agentName and
c.phone=a.agentPhone

Figure 6.1: A schema mapping system.

a similar task, but considers firms instead of independent agents. The third populates Pdb with similar data from the European data source. Unfortunately, it is in the nature of the problem of data translation that data mapped from one format to another may lose part of its semantics or may be merged with other data having different semantics.

Consider, for example, a user that is interested in estates valued at more than $500K. The following query can be executed against the portal data:

```
select * from Portal.estates where value>500K
```

The results returned to this query may not be an answer consistent with what the user wanted. It is common for American companies to not include tax in prices while European companies do. If this is the case in the above scenario, values in the result set of the query that originate from the European source will have the tax included while those originating from the American source may not. Unfortunately, there is no information in the portal schema to distinguish the two different kinds of prices. This information has been lost once the data from the two sources has been merged in the portal instance. It would be helpful if the user could specify in the query whether she is interested in data values originating from one data source or another or if the query could return, along with every data value in the answer set, the data source from which it originates.

As another example, consider a designer who notices some anomalous values in the `contacts` element in the integrated data. First, she would like to perform a query to find out from which elements the anomalous values were derived. In response to such a query, she can learn that elements from houses and agents were used to compute the anomalous values. In particular, one mapping may join `houses` and `agents` on `aid`, while another may join them on the element `owner`. She can then decide that the desired contact information in the portal should include the owner, and may correct or remove the anomalous mapping.

The above examples demonstrate a need for incorporating into queries predicates regarding how and from where the data values were derived. If the portal administrator realizes this need and has the flexibility to alter the portal schema, she can introduce some additional schema elements to keep that information. However, portal schemas are not always allowed to change, or the administrator may not be aware of subtle differences in the semantics of the values, so we may not be able to rely on an administrator to manually

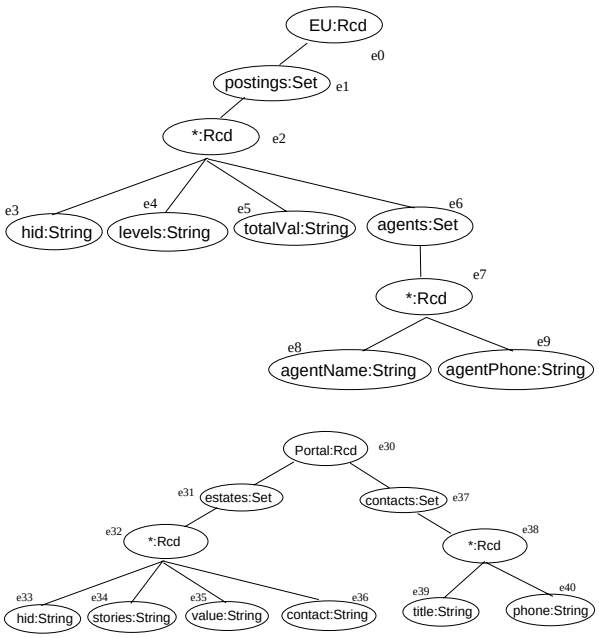


Figure 6.2: `EUdb` and `Pdb` schema graphs.

annotate the integrated data with relevant information. In this work, we propose an integrated solution to address this problem by providing a systematic way to store meta-data without relying on the integrator, and a declarative query language to use that information.

6.2 Supporting Meta-data Querying

This section provides the formal basis for realizing schema mapping systems where data, schema elements, and transformations are available for querying. It builds on the nested relational data model we have used in the previous two chapters by directly extending it to make schemas and mappings first-class citizens of the model. It then extends the query language with special operators to utilize this meta-data information. This extended query language will be referred to as MXQL.

Since our scope is now expanded and we need to work with the case where we have multiple source schemas populating a target instance, we generalize the notion of a schema

mapping system.

Definition 6.1 *A* schema mapping system *is a triple* $<\mathcal{S}_s, S_t, \mathcal{M}>$ *where* $\mathcal{S}_s$ *is a set of source schemas,* S_t *a target schema and* $\mathcal{M}$ *a set of mappings, each from a schema* $S \in \mathcal{S}_s$ *to* S_t*.*

Note that the interpretation of an element $[[e]]$ contains values of the instance that may have been generated by different mappings. We will use the notation $[[e]]^m$ to refer to the subset of $[[e]]$ that was generated through mapping m. A detailed discussion of semantics for $[[e]]^m$ and an implementation were described in Section 4. To allow schemas and mappings to be considered as data and allow them to be queried, we introduce the notion of a meta-data extended instance.

Definition 6.2 *Given a schema mapping system* $<\mathcal{S}_s, S_t, \mathcal{M}>$ *and a set containing one source instance for each schema of* $\mathcal{S}_s$ *and an instance* $\mathcal{I}_t$ *of schema* S_t *satisfying the mappings in M, if* $\mathcal{E}_t$ *represents the set of elements of schema* S_t*, a* meta-data extended instance $\mathcal{I}^M$ *is a 6-tuple* $<\mathcal{I}_t, \mathcal{S}_s, S_t, \mathcal{M}, f_{mp}, f_{el}>$ *where* f_{mp} *and* f_{el} *are total functions defined as:*

$$
\begin{array}{lll}
f_{el} : \mathcal{I}_t \rightarrow \mathcal{E}_t & s.t. & f_{el}(v) = e, \text{ where } v \in [[e]] \\
f_{mp} : \mathcal{I}_t \rightarrow \mathcal{M} & s.t. & f_{mp}(v) = \{m \mid m \in \mathcal{M} \ \wedge \ v \in [[f_{el}(v)]]^m\}
\end{array}
$$

Intuitively, function f_{el} associates each value in the target instance $\mathcal{I}_t$ of the schema mapping system with the schema element in the interpretation to which it belongs. (Recall that in our model each value is a label-value pair.) This is a functionality that in most database systems is implicit in the storage mechanism. SchemaSQL [LSS96] is an effort to overcome this limitation and allow queries on both instance values and schema elements. In the current work we go further, by incorporating into the model not only the schema elements but also the mappings that perform the data transformation.

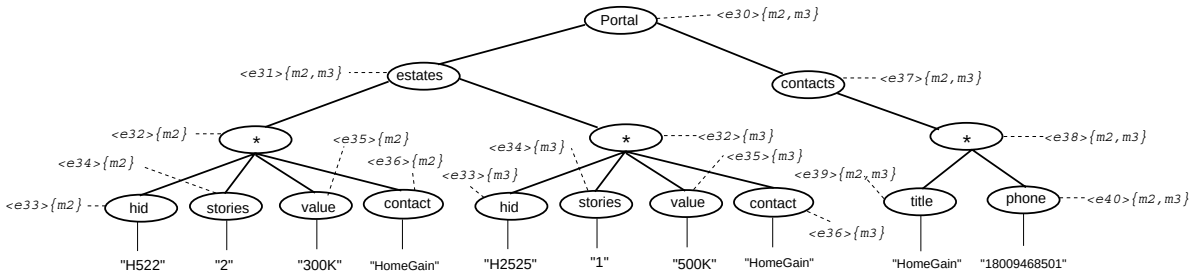


Figure 6.3: An annotated instance of the `Pdb` data source of Figure 6.1.

This is achieved through the function f_{mp}, which associates with each data value in the instance $\mathcal{I}_t$ the mappings that generated it by explicitly modeling the structure of the transformations. Note that a data value in the instance $\mathcal{I}_t$ may be generated by multiple mappings, hence, function f_{mp} returns a set. In the graph representation of an instance, the information from functions f_{el} and f_{mp} is represented through annotations on the graph nodes.

Example 6.3 *The annotations in Figure 6.3 are indicated with dotted lines. Those in angle brackets represent the element information (provided by function f_{el}), while those in curly brackets represent the mapping information (provided by function f_{mp}). For instance, the annotation of the* `title` *node indicates that it belongs to the interpretation of element e*39 *(Figure 6.2), and both mappings m_2 and m_3 have generated it.* ■

In order to be able to use databases, schema elements and mappings in queries and be able to return them in results, we extend our data model with three new atomic types: Database, Mapping and Element with domains the set of databases, mappings and elements, respectively. To access the schema element and mapping information of a value, two new operators are also introduced: @elem and @map. The first returns a value of type Element and the second a set of values of type Mapping. In particular, given an expression *exp* referring to an element e of schema S_t over a meta-data extended instance $<\mathcal{I}_t, \mathcal{S}_s, S_t, \mathcal{M}, f_{mp}, f_{el}>$ and a valuation v of *exp* over this instance, the interpretation of the two operators is: $[[exp@elem]] = f_{el}(v)$ (i.e., *exp*@elem is the schema element to which the

expression *exp* refers), and $[[exp@\mathsf{map}]] = f_{mp}(v)$ (that is, $exp@\mathsf{map}$ is the set of mappings through which the values in the interpretation of expression *exp* were derived.)

Example 6.4 *Assume that a user is interested in retrieving all the prices of the estates in the portal. To better realize the meaning of a price, i.e., if it includes tax, if it is in its original currency or has been converted, etc., the user may need to know for each price, through what transformation it was generated. This can be provided by the MXQL query:*

```
select x.estate.hid, x.estate.value, m
from   Portal.estates x, x.value@map m
```

Note that operation @map *is used in the* from *clause as a variable binding expression, since it returns a set of values.* ■

One of our main goals is to be able not only to query the mappings that generated the values in the integrated instance, but also trace back in order to find the schema elements of the sources from which these values originate or the elements on which they depend. For that, we introduce the *mapping predicate*, $<S_s{:}e_s \rightarrow m \rightarrow S_t{:}e_t>$. This Boolean mapping predicate can be used in the where clauses of the queries. It specifies certain conditions regarding the origin, the transformation and the mappings that transform the data. When these conditions are satisfied, the mapping predicate evaluates to true. Given a meta-data extended instance $<\mathcal{I}_t,\ \mathcal{S}_s,\ S_t,\ \mathcal{M},\ f_{mp},\ f_{el}>$, the interpretation of the mapping predicate is defined by:

$$[[<S_s{:}e_s \rightarrow m \rightarrow S_T{:}e_t>]] = \exists m{:}<\mathcal{E}'_s,\mathcal{E}'_t,W_c> \in \mathcal{M} \quad \text{such that}$$
$$S_T{:}<\mathcal{E}_t,f^p_t> = S_t \wedge \exists S_s{:}<\mathcal{E}_s,f^p_s> \in \mathcal{S}_s \quad \text{where} \quad \mathcal{E}'_s \subseteq \mathcal{E}_s \wedge$$
$$\mathcal{E}'_t \subseteq \mathcal{E}_t \wedge \exists e_s \in \mathcal{E}'_s \wedge \exists e_t \in \mathcal{E}'_t \wedge (e_s = e_t) \in W_c$$

Intuitively, the predicate evaluates to true if: (1) there is a mapping m that uses in the select clause of its **foreach** part an expression that refers to the element e_s of one of the

schemas in $\mathcal{S}_s$, and (2) in the select clause of its **exists** part, it uses an expression that refers to the element e_t of the target schema S_t, in a way that the values with which the mapping populates schema element e_t are the retrieved values of element e_s.

Example 6.5 *In the schema mapping system of Figure 6.1, if e_s is the* `price` *element, S_s is schema* `USdb`*, m is mapping m_1, S_t is schema* `Pdb`*, and e_t is the* `value` *element, then the predicate $<S_s{:}e_s \rightarrow m \rightarrow S_t{:}e_t>$ evaluates to true since mapping m_1 generates values for the* `value` *element by retrieving* `price` *values from the* `USdb` *database.*

The mapping predicate can be combined with the `@map` *and* `@elem` *operators to form interesting queries. Assume, for example, that a user is interested in finding estates in the* `Pdb` *database originating from the* `USdb` *data source, and specifically, those having a firm as a contact. For each such estate, the user needs to know the transformation (mapping) that generated its price. Unless the data administrator who designed the portal schema had taken special care to include that information in the schema, which is not the case in our example, this query cannot be answered since firms and individual agents have been merged under the element* `contacts` *and cannot be distinguished. Using MXQL, this requirement can be expressed as follows:*

```
select  s.hid, m
from    Portal.estates s, Portal.contacts c, c.title@map m
where   s.contact=c.title and e=c.title@elem
        and <'USdb':'US/agents/title/firm'→m→'Pdb':e>
```

The constant values `'USdb'` *and* `'Pdb'` *in the above query are of type* Database*, and the value* `'US/agents/title/firm'` *is a constant of type* Element*. The* `@map` *operator makes the* Mapping *type variable m iterate over all the mappings that generate values for the element* `title` *of the portal schema, and operator* `@elem` *makes the* Element *type variable e be the schema element* `title`*. The mapping predicate restricts variable m even further to only mappings that use* `firm` *values from the* `USdb` *to populate the* `title` *element of the*

portal. Note that variables used in the mapping predicate need not be defined in the from *clause, e.g., variable* e*. They are implicitly defined through their position in the mapping predicate. Executing the above query over the meta-data extended instance of Figure 6.3 returns tuple* $('H522', 'm_2')$*, where* $'H522'$ *is a string and* $'m_2'$ *a* Mapping *type value.* ■

MXQL queries can also be used to form queries on meta-data exclusively in order to understand the complex structure of the schemas and the way data is transformed.

Example 6.6 *Consider a user who does not completely understand what the meaning of the element* `stories` *is in the portal schema. In particular, she is wondering whether this is about the number of floors in a building or comments that users have given about specific estates. In order to find out, she asks the following query to identify from where its values originate:*

select e from
where $<db{:}e \rightarrow m \rightarrow 'Pdb':$`/Portal/estates/estate/stories`$'>$

The query returns Element *type values* `floors` *and* `levels` *as an answer, making the user realize that the element* `stories` *describes how many stories the building has. Note that, in the above query, the* from *clause is empty, since the variables used in the query are implicitly defined by their use in the mapping predicate.* ■

In many cases, for a given value in the integration, users are interested in knowing not only from where it originates, but also what other parts of the schema contain values that affect its appearance in the integration [BKT01]. To be able to answer this kind of queries, we introduce a new form of the mapping predicate, having similar syntax but using double arrows instead of single. Its interpretation is defined by:

$$[[<S_s{:}e_s \Rightarrow m \Rightarrow S_T{:}e_t>]] = \exists m{:}<\mathcal{E}'_s,\mathcal{E}'_t,W_c> \in \mathcal{M} \quad \text{such that}$$
$$S_T{:}<\mathcal{E}_t,f^p_t> = S_t \quad \wedge \quad \exists S_s{:}<\mathcal{E}_s,f^p_s> \in \mathcal{S}_s \quad \text{where} \quad \mathcal{E}'_s \subseteq \mathcal{E}_s \quad \wedge$$
$$\mathcal{E}'_t \subseteq \mathcal{E}_t \quad \wedge \quad \exists e_s \in \mathcal{E}'_s \quad \wedge \quad \exists e'_s \in \mathcal{E}'_s \quad \wedge \quad \exists e''_s \in \mathcal{E}'_s \quad \wedge \quad \exists e_t \in \mathcal{E}'_t \quad \wedge$$
$$(e_s = e'_s) \in W_c \quad \wedge \quad (e''_s = e_t) \in W_c$$

Intuitively, the predicate evaluates to true if: (1) there is a mapping m that uses in the select clause of its **exists** part the element e_t of the target schema S_T, i.e., it generates values for e_t, and (2) it uses element e_s of one of the schemas $S_s \in \mathcal{S}_s$ of the schema mapping system in its where clause. This means that the values of e_s, although they are not the values that populate element e_t, affect its population through their participation in the where clause conditions.

Example 6.7 *In the schema mapping system of Figure 6.1, the values of the element* `aid` *are not used by any of the mappings to populate an element of the portal, since the portal schema has no* `aid` *element. However, its values play an important role in the population of the target schema, since they are used to form the join between the* `house` *and* `agent` *values. Thus, element* `aid` *will be in the answer set of the query:*

```
select  c.title, e_s
from    Portal.estates s, Portal.contacts c, c.title@map m
where   s.contact=c.title and e=c.title@elem
        and <'USdb':e_s⇒m⇒'Pdb':e>
```

This information may be useful to a designer wishing to understand why certain agents appear associated with specific houses. ■

6.3 Characterization of MXQL Queries

This section provides a more specific characterization of the mapping predicates. It shows that mapping predicates describe conditions on the schema elements of the *data provenance* [CW00, BKT01].

Consider two data sources db_s and db_t with schemas S_s and S_t respectively, and a mapping m generating an instance $\mathcal{I}_t$ of S_t from instance $\mathcal{I}_s$ of S_s. Let query

$$\begin{array}{lll} q_f\text{:} & \underline{\mathsf{select}} & exp_1,\ exp_2,\ ...,\ exp_m \\ & \underline{\mathsf{from}} & P_0\ x_0,\ ...,\ P_n\ x_n \\ & \underline{\mathsf{where}} & cond_1\ \underline{\mathsf{and}}\ ...\ \underline{\mathsf{and}}\ cond_l \end{array}$$

be the **foreach** clause of mapping m and a value $v \in [[e_t]]^m$. Since $v \in [[e_t]]^m$, there is an expression exp_{e_t} in the select clause of q_f through which value v was retrieved from the source instance $\mathcal{I}_s$. Let query q'_f be one that has the same from and where clause as q_f but a select clause with all the possible valid expressions that can exist in the query. According to Buneman et al. [BKT01], although they had considered a different data model, query q'_f, with the additional condition $exp_{e_t}=v$ in its where clause, returns the *witness basis* $W_{q_f,\mathcal{I}_s}(v)$, i.e., the *why-provenance* of value v.

Definition 6.8 (Why-Provenance) *Consider two instances $\mathcal{I}_s$ and $\mathcal{I}_t$ of schemas S_s and S_t, respectively, a mapping $m : Q_s \subseteq Q_t$ from S_s to S_t, and a value v of $\mathcal{I}_t$. Assume that instance $\mathcal{I}_t$ has been generated through the mapping m from instance $\mathcal{I}_s$. If query Q_s is the query*

$$\begin{array}{lll} Q_s\text{:} & \underline{\mathsf{select}} & exp_1,\ exp_2,\ ...,\ exp_m \\ & \underline{\mathsf{from}} & P_0\ x_0,\ ...,\ P_n\ x_n \\ & \underline{\mathsf{where}} & cond_1\ \underline{\mathsf{and}}\ ...\ \underline{\mathsf{and}}\ cond_l \end{array}$$

the why-provenance *of value v through mapping m is the query*

$$\begin{array}{lll} q_f\text{:} & \underline{\mathsf{select}} & * \\ & \underline{\mathsf{from}} & P_0\ x_0,\ ...,\ P_n\ x_n \\ & \underline{\mathsf{where}} & cond_1\ \underline{\mathsf{and}}\ ...\ \underline{\mathsf{and}}\ cond_l\ \underline{\mathsf{and}}\ exp_{e_t} = v \end{array}$$

where exp_{e_t} is the expression in the select *clause of Q_s that retrieved value v.*

Query q_f with only expression exp_{e_t} in its select clause and the additional condition $exp_{e_t}=v$ in the where clause returns the *derivation basis* $\Gamma_{q_f,\mathcal{I}_s}(v)$, i.e., the *where-provenance* of value v.

Definition 6.9 (Where-Provenance) *Consider two instances $\mathcal{I}_s$ and $\mathcal{I}_t$ of schemas S_s and S_t, respectively, a mapping $m : Q_s \subseteq Q_t$ from S_s to S_t, and a value v of $\mathcal{I}_t$. Assume that instance $\mathcal{I}_t$ has been generated through the mapping m from instance $\mathcal{I}_s$. If query Q_s is the query*

$$\begin{array}{lll} Q_s\text{:} & \underline{\textsf{select}} & exp_1,\ exp_2,\ ...,\ exp_m \\ & \underline{\textsf{from}} & P_0\ x_0,\ ...,\ P_n\ x_n \\ & \underline{\textsf{where}} & cond_1\ \underline{\textsf{and}}\ ...\ \underline{\textsf{and}}\ cond_l \end{array}$$

the where-provenance *of value v through mapping m is the query*

$$\begin{array}{lll} q_f\text{:} & \underline{\textsf{select}} & exp_{e_t} \\ & \underline{\textsf{from}} & P_0\ x_0,\ ...,\ P_n\ x_n \\ & \underline{\textsf{where}} & cond_1\ \underline{\textsf{and}}\ ...\ \underline{\textsf{and}}\ cond_l\ \underline{\textsf{and}}\ exp_{e_t} = v \end{array}$$

where exp_{e_t} is the expression in the select *clause of Q_s that retrieved value v.*

We need to note here that the result of a query is not considered a simple set of values, but a set of facts, i.e., values of the instance associated with their specific positions.

Theorem 6.10 *Given two data sources db_s and db_t, and a mapping m, the mapping predicate $<db_s{:}e_s \rightarrow m \rightarrow db_t{:}e_t>$ is satisfied if and only if there is a value $v' \in [[e_s]]$ that is in the* where-provenance *of a value $v \in [[e_t]]^m$ through m.*

Proof. Assume that the mapping predicate $<db_s{:}e_s \rightarrow m \rightarrow db_t{:}e_t>$ is satisfied (evaluates to true) for some elements e_s and e_t. This means that, for a value $v \in [[e_t]]^m$, there is an expression exp_{e_t} in the select clause of q_f that generated value v, and expression exp_{e_t} refers to element e_s. For the mappings we consider here, query q_f with the additional condition $exp_{e_t}{=}v$ in its where clause and only expression exp_{e_t} in its select clause represents $\Gamma_{q_f,\mathcal{I}_s}(v)$, which means that $\Gamma_{q_f,\mathcal{I}_s}(v) \in [[e_s]]$. ∎

Let query q_f^w be a query with the same from and where clause as query q_f, but with the select clause containing all the expressions that appear in the select or the where clause of query q_f. This query defines a new form of provenance which we refer to as the *what-provenance*.

Definition 6.11 (What-Provenance) *Consider two data sources schemas S_s and S_t, a mapping $m : Q_s \subseteq Q_t$ from S_s to S_t, two instances $\mathcal{I}_s$ and $\mathcal{I}_t$ of schemas S_s and S_t, respectively, and a value v of $\mathcal{I}_t$. Assuming that instance $\mathcal{I}_t$ has been generated through the mapping m from instance $\mathcal{I}_s$, if query Q_s is the query*

select exp_1, exp_2, ..., exp_m
from P_0 x_0, ..., P_n x_n
where $cond_1$ and ... and $cond_l$

the what-provenance *of the value v is the query*

q_f: select U
from P_0 x_0, ..., P_n x_n
where $cond_1$ and ... and $cond_l$ and $exp_{e_t} = v$

where U is the set of all the expressions in the select *clause and the conditions of the* where *clause of query Q_s, and exp_{e_t} is the expression in the* select *clause of Q_s that generated value v.*

The *what-provenance* is value-based compared to the complex-structure-based (tuples or proof trees [AHV95]) *why-provenance*. It differs from why-provenance in that it does not consider the parts of the complex structures returned by the *why-provenance* that do not justify the appearance of a value in the generated instance. This difference becomes clear with the following relational example. Consider the two source schema relations $R(a, b, c)$ and $S(c, d, e)$ with contents $\{(3, 2, 3), (1, 2, 4)\}$ and $\{(4, 5, 6)\}$, respectively, and a target schema relation $T(a, b)$. Assume that the target schema relation is populated by the mapping

m: **foreach** R r, S s
with $r.c$=$s.c$
exists T t
with $t.a$=$r.a$ and $t.b$=$t.c$

As a result of a data exchange with the mapping m, the target schema relation T will be populated with only one tuple: $(1, 2)$. The *where-provenance* of value 1 in this tuple is the value of attribute a of tuple $(1, 2, 4)$ since this is from where value 1 was derived. The *why-provenance* of the same value consists of the tuples $(1, 2, 4)$ and $(4, 5, 6)$ of relations R and S, respectively, since the join of those two tuples led to tuple $(1, 2)$. However, note that attributes d and e of relation S, whatever values they may have, will not affect the result of the query. Our *what-provenance* does not include them. Attribute c on the other hand is important since the join of R and S uses c. In the specific example, the what-provenance will be the tuple $(1, 2, 4)$.

In order to characterize the relationships among the *where*, *why* and *what* provenance, we introduce the notion of *element inclusion* between queries.

Definition 6.12 (Element Inclusion) *Let q_1, q_2 be two queries. Query q_1 is* element-included *in query q_2, noted as $q_1 \sqsubseteq q_2$, if there is a total injective renaming function h from the variables of q_1 to the variables of q_2 such that the* from *and* where *clauses of the queries $h(q_1)$ and q_2 are the same, and $K_1 \subseteq K_2$ where K_1, K_2 are the ordered sets of expressions in the* select *clauses of queries $h(q_1)$ and q_2, respectively.*

Intuitively, the above definition states that if $q_1 \sqsubseteq q_2$, then for every instance $\mathcal{I}$, $q_1(\mathcal{I}) = \pi_X(q_2(\mathcal{I}))$ where π_X means projection on some set of elements X. Note that if a query q_1 is element included in query q_2, then query q_1 is dominated by q_2, but not vice versa.

It is clear from the above definition that $q_f^w \sqsubseteq q'_f$, where query q'_f was defined a few paragraphs before to represent the why-provenance. In general, if queries q_{why}, q_{where} and q_{what} represent the *why*, *where* and *what-provenance*, respectively, it holds that $q_{where} \sqsubseteq q_{what} \sqsubseteq q_{why}$.

We will show next that the mapping predicate with double arrows relates to the

what-provenance.

Theorem 6.13 *Given two data sources db_s and db_t, and a mapping m, the mapping predicate $<db_s{:}e_s \Rightarrow m \Rightarrow db_t{:}e_t>$ is satisfied if and only if a value $v \in [[e_s]]$ is in the* what-provenance *of a value $v' \in [[e_t]]^m$ through mapping m.*

Proof. If the mapping predicate $<db_s{:}e_s \Rightarrow m \Rightarrow db_t{:}e_t>$ is satisfied for some elements e_s and e_t, then, for an instance value $v \in [[e_t]]^m$, there is an expression exp_j in the select or the where clause of q_f that refers to the schema element e_s. By definition, expression exp_j will be in the what-provenance select clause, hence, the interpretation of element $[e_s]$ is in the what-provenance. ∎

An interesting observation here is that for a mapping m, the set of schema elements e_s, for which the double-arrow predicate is satisfied, is a subset of the set of schema elements to whose interpretation the *why-provenance* of the instance values $v \in [[e_t]]^m$ conforms.

6.4 Implementing MXQL

This section describes our implementation of the MXQL query language using existing technology. Schemas and mappings, in order to be queried and returned in answer sets as regular data, need to be stored. Although there may be many different methodologies targeting different applications with different requirements, we propose a storage schema we believe is quite generic, and we explain how MXQL queries can be executed by exploiting that storage schema.

6.4.1 Meta-data Physical Storage Schema

Figure 6.4 presents a number of Set of Rcd[...] types (represented as relations for notational simplicity) used to store meta-data information. Recall that a relation in our model is a schema root R of type Set of Rcd$[a_1{:}\tau_1, ..., a_n{:}\tau_n]$ where every type τ_k is an atomic type.

Db(*name*)
Element(*eid*, *name*, *type*, *parent*, *db*)
Query(*qid*)
Binding(*bid*, *qid*, *eid*, *prev*)
Condition(*qid*, *bid*, *eid*, *op*, *bid*2, *eid*2)
Mapping(*mid*, $forQ$, $conQ$)
Correspondence(*mid*, $forBid$, $forEid$, $conBid$, $conEid$)

Figure 6.4: Meta-data storage schema.

The *Element* relation encodes the graph representation of a schema using the edge approach [FK99a]. Each tuple corresponds to a node, i.e., a member of the set $\mathcal{E}$ of schema $<\mathcal{E},f^p>$. Attribute *name* records its label and *type* specifies its type. Function f^p of schema $<\mathcal{E},f^p>$ is implemented through the *parent* attribute that specifies the parent node. Finally, attribute *db* refers to the name of the data source.

Three relations are used to model queries: *Query*, *Binding* and *Condition*. Each query gets a unique identifier that is recorded in relation *Query*. Relation *Binding* records the from clause of each query as a list of bindings. Each tuple represents a binding and has a unique identifier (*bid*) within a query (*qid*). For the binding $P_i\ x_i$, variable x_i becomes the binding identifier. Expression P_i is represented by two parts: the variable or schema root with which it starts and the schema element to which it refers. The first is encoded in the attribute *prev*, referencing the corresponding binding. The second is encoded in the attribute *eid*, referencing the corresponding element identifier in the *Element* relation. Since queries have no bindings for schema roots, implicit bindings are introduced for each schema root used in the query. These bindings have the *prev* attribute set to null. The where clause of a query is modeled in the *Condition* relation, where each tuple represents a condition. Conditions are of the form $expr_1 \theta expr_2$. Each of the expressions $expr_1$ and $expr_2$ is encoded similarly to expression P_i, that is, by specifying the variable or schema root with which it starts and the schema element to which it refers. Attributes *bid* and *eid* encode expression $expr_1$, and attributes *bid*2 and *eid*2 encode $expr_2$. Attribute *op* designates the operator θ.

The *Mapping* relation is used to encode mappings. Attribute *mid* is a unique identifier, while attributes $forQ$ and *conQ* specify the queries in the **foreach** and **exists** part respectively. The expressions in the select clause of the two queries of the mapping are encoded in the *Correspondence* relation in a way similar to relation *Conditions*, but with the *op* attribute taken by default to be the "=". In general, the *Correspondence* and *Condition* relations are used to encode the W_c set of a mapping $<\mathcal{E}_s,\mathcal{E}_t,W_c>$, while the sets $\mathcal{E}_s$ and $\mathcal{E}_t$ are encoded in the relation *Binding* in conjunction with the relations *Query*, *Mapping* and *Element*.

Example 6.14 *Figure 6.5 shows an instance of the meta-data schema that encodes the* `Pdb` *and* `EUdb` *data source schemas and mapping* m_3 *of Figure 3.1.* ■

Since mappings are expressed as inter-schema constraints, one could use the same mapping storage mechanism to also store intra-schema constraints, e.g., foreign keys, and use them in queries.

6.4.2 Annotating the Instance Data

Annotations are not part of the nested relational model and are not supported directly by the nested relational queries. Hence, to use annotations in queries, we implement functions $getElAnnot(v)$ and $getMapAnnot(v)$, which return respectively the element annotation and the set of mapping annotations of a value v in the meta-data extended instance. The advantage of using functions is implementation independence. In an XML data repository, for example, annotations may be implemented as attributes on elements, while in a relational repository as auxiliary tables.

The population of a schema through a set of GLAV mappings, as described in Chapter 4, does not support annotations [PVM+02b]. To cope with this problem, we first rewrite each mapping by augmenting its **with** clause in order to explicitly generate the annotations. In particular, given a mapping m, for every expression *expr* in the **with**

Element

eid	name	type	parent	db
e0	EU	Rcd	–	EUdb
e1	postings	Set	e0	EUdb
e2	*	Rcd	e1	EUdb
e3	hid	Str	e2	EUdb
e4	levels	Str	e2	EUdb
e5	totalVal	Str	e2	EUdb
e6	agents	Set	e2	EUdb
e7	*	Rcd	e6	EUdb
e8	agentName	Str	e7	EUdb
e9	agentPhone	Str	e7	EUdb
e30	Portal	Rcd	–	Pdb
e31	estates	Set	e30	Pdb
e32	*	Rcd	e31	Pdb
e33	hid	Str	e32	Pdb
e34	stories	Str	e32	Pdb
e35	value	Str	e32	Pdb
e36	contact	Str	e32	Pdb
e37	contacts	Set	e30	Pdb
e38	*	Rcd	e37	Pdb
e39	title	Str	e38	Pdb
e40	phone	Str	e38	Pdb

Query

qid
q0
q1

Mapping

mid	forQ	conQ
m3	q0	q1

Condition

qid	bid	eid	op	bid2	eid2
1	e	e36	=	c	e39

Correspondence

mid	forBid	forEid	conBid	conEid
m3	p	e3	e	e33
m3	p	e4	e	e34
m3	p	e5	e	e35
m3	a	e8	c	e39
m3	a	e9	c	e40

Binding

bid	qid	eid	prev
r_1	q0	e0	–
p	q0	e1	r_1
a	q0	e6	p
r_2	q1	e30	–
e	q1	e31	r_2
c	q1	e37	r_2

Figure 6.5: Meta-data storage implementation.

clause referring to element e of the target schema, expressions $getElAnnot(expr) =' e'$ and $getMapAnnot(expr) =' m'$ are also appended to the **with** clause. The rewritten mappings can then be executed as described in Chapter 4 to generate the annotated instance.

Example 6.15 *Mapping m_2 of Figure 3.1 is rewritten to:*

m_2: **foreach** `US.houses` h, `US.agents` a, a.`title`→`firm` f
  where h.`aid`=a.`aid`
exists `Portal.estates` e, `Portal.contacts` c
  where e.`contact`=c.`title`
with e.`hid`=h.`hid` and e.`stories`=h.`floors` and
  e.`value`=h.`price` and c.`title`=f and

c.phone=a.phone
$getElAnot(e.\texttt{hid})='/Portal/estates/hid'$ and $getMapAnnot(e.\texttt{hid})='m_2'$,
$getElAnot(e.\texttt{stories})='/Portal/estates/stories'$ and
$getMapAnnot(e.\texttt{stories})='m_2'$, ...

■

6.4.3 Translating MXQL queries

This section describes how an MXQL query is translated into a query that can be executed over a meta-data extended instance. The first step is to translate every e@map and e@elem operation in the query to a function call of $getMapAnnot(e)$ or $getElAnnot(e)$, respectively. Mapping predicates are processed next. If a mapping predicate contains constants, each is replaced by a new variable, and a condition is added in the where clause requiring that the variable be equal to the constant value.

Example 6.16 *Applying these steps to the query in Example 6.7 gives:*

select $s.\texttt{hid}$, m
from Portal.estates s, Portal.contacts c,
$getMapAnnot(c.\texttt{title})$ m_v
where $s.\texttt{contact}=c.\texttt{title}$ and $<db{:}e{\rightarrow}m{\rightarrow}db_2{:}e_2>$
and $m=m_v$ and $e='\texttt{US/agents/title/firm}'$
and $e_2=getElAnnot(c.\texttt{title})$ and $db='\texttt{US}'$
and $db_2='\texttt{Pdb}'$

■

The bindings of the variables in the mapping predicates are generated next. In particular, for the predicate $<db{:}e{\rightarrow}m{\rightarrow}db_2{:}e_2>$, variables e and e_2 are bound to relation *Element* and m to relation *Mapping*. Existing references to these variables in the select and the where clauses are then replaced by references to the identifier attribute of the

corresponding table. For instance, since variable e is bound to *Element*, every expression involving variable e is replaced by $e.eid$. Variables db and db_2 are finally replaced by expression $e.db$ and $e_2.db$, respectively.

Example 6.17 *Applying the above steps to the query of Example 6.16 results in:*

select s.hid, $m.mid$
from Portal.estates s, Portal.contacts c, $Mapping\ m$,
$getMapAnnot(c.\texttt{title})\ v_n$, $Element\ e$, $Element\ e_2$
where $s.\texttt{contact}{=}c.\texttt{title}$ and $<e.db{:}e{\rightarrow}m{\rightarrow}e_2.db{:}e_2>$
and $m_v{=}m.mid$ and $e.eid{=}'\texttt{US/agents/title/firm}'$
and $getElAnnot(c.\texttt{title}){=}e_2.eid$ and $e.db{=}'\texttt{US}'$
and $e_2.db{=}'\texttt{Pdb}'$

■

The last step is to specify how the variables of the mapping predicate relate to one another. The relationship between the element e and the mapping m depends on the mapping predicate. If it is a single arrow predicate, it means that the element e is used in the select clause of the **foreach** part of the mapping m. This translates to the conditions $e.eid{=}o.forEid$ and $o.mid{=}m.mid$ where o is a new variable that binds to *Correspondence*. If we have a double-arrow predicate, then the case is similar, but the requirement for e is to be used either in the select clause as before or in the where clause. In the latter case, this translates to the existence of a corresponding entry in the *Condition* relation. The association between mapping m and element e_2 is analogous. The mapping predicates are finally removed.

Example 6.18 *The query of Example 6.17 becomes:*

select s.hid, $m.mid$
from Portal.estates s, Portal.contacts c,
$getMapAnnot(c.\texttt{title})\ m_v$, $Element\ e$
$Mapping\ m$, $Element\ e_2$, $Correspondence\ o$
where $s.\texttt{contact}{=}c.\texttt{title}$ and $m_v{=}m.mid$
and $getElAnnot(c.\texttt{title}){=}e_2.eid$ and $e.db{=}'\texttt{US}'$
and $e.eid{=}'\texttt{US/agents/title/firm}'$ and $o.mid{=}m.mid$
and $o.forEid{=}e.eid$ and $o.conEid{=}e_2.eid$ and $e_2.db{=}'\texttt{Pdb}'$

■

The advantage of using MXQL queries over the rewritten form we have just described is that the user does not need to be aware of the details of the meta-data storage schema. Instead, she only has to declaratively specify her requirements in the query. Another advantage is that the storage method can be modified without altering the application queries. Since the queries are most of the time small enough to fit in memory, the translation of the MXQL queries raises no real performance issues.

6.5 Experience

As an application scenario for our framework, we used the generation of a real estate portal that integrates and materializes data from five popular real estate Web sites (Yahoo, NK Realtors, Winderemere, Westfall and Homeseekers) with an average schema size of 55 elements. The information extracted from the Web sites consists of a total of 14.3MB of XML data (10,000 real estate listings). This information was mapped through a number of nested relational mappings to an integrated schema having 135 elements. Execution of the mappings generated an integrated instance of 14.5MB, which is slightly larger than the sum of the total size of the instance data. It is larger because many pieces of information from the data sources were represented more than once in the portal instance. For example, the contact phone number from the Yahoo data source was mapped to both the `business` and the `home` phone in the integrated schema.

The mappings were given to a pre-processor that rewrote them in order to generate annotations (as described in Section 6.4.2). The re-written mappings were executed on the sources and generated the integrated instance. The new annotated instance was 3 MB larger than the instance without the annotations. This was expected since every XML element carries its annotations, which are represented as XML attributes. Exploiting the

fact that the generation methodology (Chapter 4) produces a data instance in Partition Normal Form [AB86], we were able to avoid storing mapping annotations on the children of complex type value elements since they are the same as the annotations of their parents. This reduced the space overhead of the 3MB to only 0.8MB (i.e., 5.5% of the size of the integrated instance). The schemas and the mappings were encoded as described in Section 6.4.1, and were included in the annotated instance. With this inclusion, the annotated instance became only 0.3MB larger. We performed a number of experiments with different sizes of source data, and the results showed that the increase in the space of the integrated instance, due to the annotations, was approximatelly 5.5% in all the cases. The increase in space caused by the storing of schemas and the mappings was approximately 0.3MB. The real estate sites we used had very few overlapping data, i.e., few entries appeared more than one data sources. We mapped parts of the data of Windermer to Westfall and Homeseekers, and parts of the Yahoo data to NK Realtors. As a result, different information about the same real estate entry was now appearing in different sources. We generated the integrated instance as before, and we noticed that the extra space needed by the annotations went down to 4.9%. This means that the space overhead in the annotated instance is less if the sources have overlapping information. Furthermore, we expect that the annotation space overhead should decrease even further if the number of nested sets of the integrated schemas increases. In the future, we plan to perform more experiments and a deeper study of how the extra space required caused in the integrated instance for the annotations and the meta-data is affected by the size and the kind of the source data.

We executed a number of MXQL queries over the annotated instance, but we noticed no significant execution time increase. Furthermore, MXQL queries helped identify the meaning of some elements in the integrated schemas in a way similar to the one described in Example 6.6. In addition, they helped detect ill-defined mappings. For example, we noticed that the sub-element `housesInNeighborhood` of a `house` element, in some

cases, contained houses that were from different states. In investigating the problem, we executed the following MXQL query

select db, e from where $<db{:}e \Rightarrow m \Rightarrow' Portal':e>$ and
$e='/Portal/house/housesInNeighborhood'$

For the Homeseekers data source, it returned two elements only: the `hid` and the `neighborhood`. Indeed, the Homeseekers mapping was computing the neighboring houses by performing a self-join only on the element `neighborhood`. Unfortunately, neighborhoods with the same name could appear in different states, thus generating the misleading data. When the mapping was updated to join on city, state, and neighborhood, the problem was corrected.

MXQL helped us judge the accuracy of the mappings. We noticed, for example, that for some houses the high, middle and elementary school districts were the same, although for other houses in the same area they were not. In querying the mappings of the latter values, we noticed that, for the mappings originating from the Realtors data source, all three elements were retrieving their values from a single element `schoolDistrict`, since the Realtors source was not separating elementary, middle and high school districts.

These experiments demonstrate the importance of detecting and managing the schema level origin of the data, as well as the transformations that have generated this data. The experiments have also indicated that we can achieve this functionality with a relatively very small cost in additional storage space and negligible additional query execution time.

Chapter 7

Conclusion

Managing mappings is a critical task in numerous data management applications. Manual mapping management is laborious, time-consuming and error prone. Given the rapid proliferation and the growing size of meta-data in the current Web environment, automatic meta-data management techniques become ever more important. Unfortunately, full automation is hard to achieve. However, it is important and feasible to develop techniques and tools to assist data administrators in managing the mappings as efficiently as possible. This dissertation has contributed to both understanding the mapping management problem and developing mapping management tools. In this chapter, we recap the key contributions of the dissertation and discuss directions for future research.

7.1 Key Contributions

The area of meta-data management is a broad area that has recently received considerable attention. This dissertation makes three main contributions in that area, in particular, in schema mapping generation [PVM+02b, AFF+02, PHV+02, PVM+02a], mapping maintenance [VMP03b, VMP04, VMP03a, VMPM04], and meta-data querying [VMM05].

7.1.1 Mapping Generation

We presented a framework that takes as input the output of a (manual or automatic) schema matching process, which is a high-level description of how the elements of two schemas correspond, and turns this high-level description into a set of mappings that describe how the data described by the two schemas relate to each other. The data model we used is one that can express relational schemas and nested structures like those met in XML schemas. The mappings that are generated are from a broad class of mappings, called GLAV mappings. GLAV mappings naturally include the GAV and LAV mappings that are used extensively in data integration and data exchange. The advantage of our approach is that it uses information from the schema structure and semantics (schema constraints) in order to infer the semantics of the high-level correspondences and produce the mappings. The generated mappings are guaranteed not to violate any of the constraints of the schemas. In practice, data administrators often generate mappings without considering the constraints of the target schema. If the mappings are to be used for data exchange, the they may generate data for the target schema that violate the schema constraints. Due to this inconsistency, the data may fail to be inserted in the target schema, and the data administrator will have to modify them. Our approach, helps the data administrators save significant time by ensuring at the mapping generation time that the mappings will not violate the target schema constraints. Hence, the population of the target schema will not fail due to constraint violation.

Furthermore, for the data exchange case, the mappings need to be translated to actual queries that translate data conforming to the first schema to data conforming to the second. During that process, some elements of the second schema may be required, but the mappings may not specify from where their values should be derived. We have developed a process that can generate values for such elements in a way that its consistent to the schema structure and constraints. Our framework can produce queries in SQL, XQuery, or XSLT.

We have implemented the mapping and query generation framework in the tool Clio. We have described the implementation and the experiments we performed in order to study the effectiveness of such a tool.

7.1.2 Mapping Maintenance

For dynamic environments like the Web, where data sources may change their schemas at any time, we have developed a framework for mapping maintenance. The framework can continuously monitor the mappings that have been defined between two schemas, and whenever there is a schema change, it can automatically detect what mappings are affected by that change. We have also developed a mapping adaptation algorithm that can generate rewritings of the affected mappings based on the change that took place and the previous mappings.

One of the basic principles of our approach is to preserve the semantics of the initial mappings as much as possible. No changes are performed to the mappings during the rewriting process unless it is necessary to do so. Our framework can adapt mappings in response to a variety of schema changes that involve not only structural modifications, which may render the mappings invalid, but also semantic modifications, which may render the mappings semantically inconsistent.

The problem of finding affected mappings and updating them is a new problem, and we are the first to investigate it in this form. We call it the mapping adaptation problem to distinguish it from the the similar problems of schema evolution, view maintenance, and view adaptation.

We have implemented this framework in a tool called ToMAS, and we have performed a number of experiments. In these experiments, we compared the incremental maintenance approach with the approach of a mapping generation tool, like Clio, that rebuilds the mappings from scratch when the schemas are modified. The experiments have shown that the incremental approach is preferable when the changes are not revolutionary, i.e.,

they do not change the schema dramatically.

7.1.3 Meta-data Querying

We have considered data provenance at the schema and mapping level. We considered how we can declaratively describe the origin of data and its transformations. In particular, we considered how to answer questions such as "what schema elements in the source(s) contributed to this value that appears in the integration?", "through what transformations or mappings was this value derived?" or "what data elements were derived through transformations sharing a certain property?". Towards this end, we elevated schemas and mappings to first-class citizens of the repository and the query language. We presented a framework for storing schemas and mappings in a data repository and associating them with the actual data values. We also developed an extended query language called MXQL, which can exploit that storage mechanism to allow meta-data to be queried as regular data. We described the implementation and our experience with applying this framework in a real life scenario, and showed its feasibility.

7.2 Future Directions

We are planning to continue working towards the support of systematic and efficient data translation. Our goal is to continue building systems and tools that will support data administrators in the laborious task of data translation and meta-data management. This research area is in its infancy, and substantial work remains to be done towards the goal of achieving a comprehensive meta-data management solution. In what follows, we discuss several directions for future work.

7.2.1 Multisets

The data model we have used in this dissertation is based on set semantics. An interesting research issue is what happens when we have multi-set semantics. The mapping generation algorithm may not be affected, but the data translation process is heavily dependent on the assumption of the set semantics, and may have to be modified for multi-sets.

7.2.2 Order

The order of the data has not been taken into consideration. In XML data, order plays an important role. The mapping language may have to be enhanced with the appropriate constructs in order to support queries that involve order specifications.

7.2.3 More query language features

In addition to order, there are several other language features that require further investigation. For example, the mappings we have considered here are based on conjunctive queries. Conjunctive queries cover a great part of the queries that are met in practice and in the research literature. However there are cases where other kinds of queries are needed. For instance, data warehouse applications require queries that involve aggregation, i.e., min, max, avg, etc. Another example is the Web interfaces where queries with inequalities, i.e., $=$, $\leq$, $\geq$, are very common. Supporting mappings that involve inequalities or aggregate functions may require our methods to be revisited.

7.2.4 Semantic information.

The mapping generation and maintenance framework exploits information that is encoded in the schema structure and constraints. However, the schema structure and the schema constraints are not expressive enough to describe all the semantic information of the data

they represent. It is true that, in many cases, different semantic information is modeled in the same way in the schema. For example, an isA relationship may be modeled as a foreign key constraint, and the same modeling approach may be used to model a many-to-one relationship between two concepts. Investigating how a semantic model can help in improving the mapping generation and maintenance is one of the challenges we are planning to work on in the future. Results from such research will be useful for areas like the Semantic Web.

7.2.5 Bi-directional mappings

In this dissertation, we assumed that we had two schemas and an instance of the first schema only. In many application scenarios, like the peer-to-peer systems, we have a model where both sources are populated with data, and data may flow in both directions: from the first schema to the second, and from the second to the first. It is a research challenge to investigate how mappings can describe bidirectional data exchange and how such mappings interact.

Bibliography

[AB86] S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer and System Sciences*, 33:361–393, December 1986.

[ACM97] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Translation for Heterogeneous Data. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 351–363, Delphi, Greece, 1997.

[AD98] S. Abiteboul and O. M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 254–263, 1998.

[AFF+02] P. Andritsos, R. Fagin, A. Fuxman, L. Haas, M. Hernndez, H. Ho, A. Kementsietsidis, R. J. Miller, F. Nauman, L. Popa, Y. Velegrakis, C. Vilarem, and L. Yan. Schema Management. *IEEE Data Engineering Bulletin*, 25(3), September 2002.

[AHCM94] R. Abu-Hamdeh, J.R. Cordy, and T.P. Martin. Schema Translation Using Structural Transformation. In *IBM CAS Conference (CASCON)*, pages 202–215, 1994.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKK+03] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion Project: From Data Integration To Data Coordination. *SIGMOD Record*, 32(3):53–58, 2003.

[AS03] K. Anyanwu and A. P. Sheth. ρ-Queries: Enabling Querying for Semantic Associations on the Semantic Web. In *Proceedings of the International WWW Conference*, pages 690–699, 2003.

[ASD+91] R. Ahmed, P. De Smedt, W. Du, W. Kent, M. A. Ketabchi, W. A. Litwin, A. Rafii, and M. Shan. The Pegasus Heterogeneous Multidatabase System. *Computer*, 24(12):19–27, December 1991.

[AT95] P. Atzeni and R. Torlone. Schema Translation between Heterogeneous Data Models in a Lattice Framework. In *Proceedings of the IFIP TC-2 Working Coneference on Data Semantics (DS-6)*, pages 345–364, May 1995.

[AT97] P. Atzeni and R. Torlone. MDM: A Multiple-Data Model Tool for the Management of Heterogeneous Database Schemes. In *ACM SIGMOD Conference*, pages 528–531, 1997.

[BA00] A. Bairoch and R. Apweiler. The SWISS-PROT Protein Sequence Database and its Supplement TrEMBL in 2000. *Nucleic Acids Res*, 28(1):45–48, 2000.

[BCF+03] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both Types and Constraints in Data Integration. In *ACM SIGMOD Conference*, pages 277–288, 2003.

[BCTV04] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 900–911, 2004.

[BDK92] François Bancilhon, Claude Delobel, and Paris Kanellakis. *Implementing an Object-Oriented database system: The story of O_2*. Morgan Kaufmann, 1992.

[BDM02] S. Bowers, L. Delcambre, and D. Maier. Superimposed Schematics: Introducing E-R Structure for In-Situ Information Selections. In *ER*, 2002.

[BDNS94] M. Buchheit, F. Donini, W. Nutt, and A. Schaerf. Terminological Systems Revisited: Terminology=Schema + Views. In *Proceedings of KI'94 Workshop KRDB'94*, Saarbrcken, Germany,, September 20-22, 1994.

[Ber03] P. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the Conferenence on Innovative Data Systems Research (CIDR)*, pages 209–220, January 2003.

[BFH+02] P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. LegoDB: Customizing Relational Storage for XML Documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1091–1094, 2002.

[BGF+97] S. Bressan, C. H. Goh, K. Fynn, M. J. Jakobisiak, K. Hussein, H. B. Kon, T. Lee, S. E. Madnick, T. Pena, J. Qu, A. W. Shum, and M. Siegel. The COntext INterchange Mediator Prototype. In *ACM SIGMOD Conference*, pages 525–527, 1997.

[BHL83] E. Bertino, L. M. Haas, and B. G. Lindsay. View Management in Distributed Data Base Systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 376–378, 1983.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-oriented Databases. In *ACM SIGMOD Conference*, pages 311–322, May 1987.

[BKL+04] M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, 2004.

[BKT01] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *Proceedings of the International Conference in Database Theory (ICDT)*, 2001.

[BKT02] P. Buneman, S. Khanna, and W. Tan. On Propagation and Deletion of Annotations Through Views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[BLP00] P. Bernstein, A. Levy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Record*, 29(4):55–63, December 2000.

[BLT86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD Conference*, pages 61–71, May 1986.

[BM99] C. Beeri and T. Milo. Schemas for Intergration and Translation of Structured and Semi-structured Data. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 296–313, 1999.

[Bre96] P. Breche. Advanced Primitives for Changing Schemas of Object Databases. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, May 1996.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 4–19, 1998.

[CCDF+01] D. Chamberlin, J. Clark, J. Robie D. Florescu, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. W3C Working Draft, 2001. http://www.w3.org/TR/xquery/.

[CCGL02] A. Cali, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data Integration Under Integrity Constraints. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 262–279, 2002.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD Conference*, pages 177–188, June 1–4 1998.

[CGGL04] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[CGM97] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *ACM SIGMOD Conference*, pages 26–37, 1997.

[CHS+95] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Research Issues in Data Engineering*, pages 124–131, Los Alamitos, Ca., USA, March 1995. IEEE Computer Society Press.

[CJR98] K. T. Claypool, J. Jin, and E. A. Rundensteiner. SERF: Schema Evolution Through an Extensible Re-usable and Flexible Framework. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 314–321, November 3–7 1998.

[Cla99] J. Clark. XSL Transformations (XSLT). W3C Recommendation, 1999. http://www.w3.org/TR/xslt.

[Cod79] E. F. Codd. Extending the Relational Model to Capture More Meaning. *ACM Transactions on DB Systems*, 4(4):394–434, December 1979.

[Cod90] E. F. Codd. *The Relational Model for Database Management - Version 2*. Addison-Wesley, Reading, 1990.

[Con98] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/TR/REC-xml, February 1998.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 277–289, September 1991.

[CW00] Y. Cui and J. Widom. Practical Lineage Tracing in Data Warehouses. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2000.

[CW03] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. *International Journal on Very Large Data Bases*, 12(1):41–58, 2003.

[DD99] R. Domenig and K. R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 28(3):63–72, 1999.

[DDH01] A. Doan, P. Domingos, and A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. In *ACM SIGMOD Conference*, pages 509–520, 2001.

[DFF+98] A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submssion to the World Wide Web Consortium, August 1998. http://www.w3.org/TR/NOTE-xml-ql.

[DK97] S. Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 55–66, April 1997.

[DR02] H. H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 610–621, 2002.

[FK99a] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, May 1999.

[FK99b] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FKMP03a] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering, 2003.

[FKMP03b] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 207–224, 2003.

[FKP03] R. Fagin, P. G. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 90–101, 2003.

[FKPT04] R. Fagin, P. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2004.

[FL96] F. Ferrandina and S. Lautemann. An Integrated Approach to Schema Evolution for Object Databases. In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS)*, pages 280–294, December 1996.

[FLM99] M. Friedman, A. Levy, and T. Millstein. Navigational Plans for Data Integration. In *Proceddings of the National Conference on Artificial Intelligence (AAAI)*, pages 67–73, 1999.

[Gal99] A. Gal. Semantic Interoperability in Information Services. Experience with CoopWARE. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(1):68–75, Mar 1999.

[GLS95] M. Gyssens, L. Lakshmanam, and I. N. Subramanian. Tables as a Paradigm for Querying and Restructuring. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 93–103, 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[GM99] G. Grahne and A. O. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 332–347, 1999.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[GMR95] A. Gupta, I. Mumick, and K. Ross. Adapting Materialized Views After Redefinition. In *ACM SIGMOD Conference*, pages 211–222, 1995.

[HC03] B. He and K. C. Chang. Statistical Schema Matching across Web Query Interfaces. In *ACM SIGMOD Conference*, pages 217–228, 2003.

[HIST03] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema Mediation in Peer Data Management Systems. In *Proceedings of International Conference on Data Engineering (ICDE)*, page 505, 2003.

[HQGW93] N. I. Hachem, K. Qiu, M. A. Gennert, and M. O. Ward. Managing Derived Data in the Gaea Scientific DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1993.

[Hul86] R. Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal of Computing*, 3(15):856–886, 1986.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 455–468, 1990.

[KAM03] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *ACM SIGMOD Conference*, pages 325–336, 2003.

[KKPS01] J. Kaha, M. Koivunen, E. Prud'Hommeaux, and R. R. Swick. Annotea: An Open RDF Infrastructure for Shared Web Annotations. In *Proceedings of the International WWW Conference*, pages 623–632, May 2001.

[KLSS95] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, 1995.

[KMRS92] M. Kantola, H. Mannila, K.-J. Rih, and H. Siirtola. Discovering Functional and Inclusion Dependencies in Relational Databases. *International Journal of Intelligent Systems*, 7(7):591–607, September 1992.

[KN03] J. Kang and J. F. Naughton. On Schema Matching with Opaque Column Names and Data Values. In *ACM SIGMOD Conference*, pages 205–216, 2003.

[KR99] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *ACM SIGMOD Conference*, pages 371–382, 1999.

[Len02] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[Ler00] B. S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM Transactions on Database Systems (TODS)*, 25(1):83–127, March 2000.

[LGM96] W. J. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 63–74, 1996.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 95–104. ACM, May 1995.

[LNR02] A. J. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE Transactions On Knowledge and Data Engineering*, 14(5):931–954, 2002.

[LRO96] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 251–262, 1996.

[LRS02] Q. Lv, S. Ratnasamy, and S. Shenker. Can Heterogeneity make Gnutella Scalable? In *Proceedings of International Workshop on Peer-to-Peer Systems*, pages 94–103, 2002.

[LSS96] L. V.S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL: A Language for Interoperability in Multiple Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, 1990.

[MBR01] J. Madhavan, P. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 49–58, 2001.

[MD96] M. K. Mohania and G. Dong. Algorithms for Adapting Materialised Views in Data Warehouses. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications*, pages 309–316, December 1996.

[MGMR02] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 117–128, 2002.

[MH03] J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 572–558, 2003.

[MHH00] R. J. Miller, L. M. Haas, and M. Hernandez. Schema Mapping as Query Discovery. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, September 2000.

[MHH+01] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIGMOD Record*, 30(1), 2001.

[MHI+95] H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 120–133, 24–27 August 1993.

[MMS79] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems (TODS)*, 4(4):455–469, 1979.

[MP02] P. McBrien and A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 484–499, 2002.

[MP03] P. McBrien and A. Poulovassilis. Data Integration by Bi-Directional Schema Transformation Rules. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 227–238, 2003.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *ACM SIGMOD Conference*, pages 100–111, May 13–15 1997.

[MRB03] S. Melnik, E. Rahm, and P. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *ACM SIGMOD Conference*, pages 193–204, June 2003.

[MRT98] G. Mihaila, L. Raschid, and A. Tomasic. Equal Time for Data on the Internet with WebSemantics. In *International Conference on Extending Database Technology (EDBT)*, pages 87–101, March 1998.

[MU83] D. Maier and J. D. Ullman. Maximal Objects and the Semantics of Universal Relation Databases. *ACM Transactions on Database Systems (TODS)*, 8(1):1–14, 1983.

[MUV84] D. Maier, J. D. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM Transactions on Database Systems (TODS)*, 9(2):283–308, June 1984.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 122–133, 1998.

[Nav80] S. B. Navathe. Schema Analysis for Database Restructuring. *ACM Transactions on Database Systems (TODS)*, 5(2):157–184, 1980.

[NHT+02] F. Naumann, C.T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute Classification Using Feature Analysis. In *Proceedings of International Conference on Data Engineering (ICDE)*, page 271, 2002.

[PHV+02] L. Popa, M. A. Hernández, Y. Velegrakis, R. J. Miller, F. Naumann, and H. Ho. Mapping XML and Relational Schemas with CLIO, *System Demonstration.* In *Proceedings of International Conference on Data Engineering (ICDE)*, 2002.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Pop00] L. Popa. *Object/Relational Query Optimization with Chase and Backchase.* PhD thesis, University of Pennsylvania, 2000.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 802–813, 2002.

[PT99] L. Popa and V. Tannen. An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. In *Proceedings of the International Conference in Database Theory (ICDT)*, pages 39–57, 1999.

[PVM+02a] L. Popa, Y. Velegrakis, R. J. Miller, M. Hernández, and R. Fagin. Translating Web Data. Technical Report CSRG-441, Un. of Toronto, Dep of Comp. Sc., February 2002. ftp://ftp.cs.toronto.edu/cs/ftp/pub/reports/csri/441.

[PVM+02b] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 598–609, August 2002.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Proceedings of International Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[RAJB+00] J. F. Roddick, L. Al-Jadir, L. E. Bertossi, M. Dumas, F. Estrella, H. Gregersen, K. Hornsby, J. Lufter, F. Mandreoli, T. Mannisto, E. Mayol, and L. Wedemeijer. Evolution and change in data management - issues and directions. *SIGMOD Record*, pages 21–25, 2000.

[RB01] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *International Journal on Very Large Data Bases*, 10(4):334–350, 2001.

[RP03] P. A. Bernstein R. Pottinger. Merging Models Based on Given Correspondences. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 826–873, 2003.

[RR99] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In *Management of Heterogeneous and Autonomous Database Systems*, pages 119–155. Morgan Kaufmann, 1999.

[SHT+77] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A Data EXtraction, Processing and REstructuring System. *ACM Transactions on Database Systems (TODS)*, 2(2):134–174, 1977.

[SMLN+] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*. To Appear in IEEE/ACM Transactions on Networking.

[SP94] S. Spaccapietra and C. Parent. View Integration : A Step Forward in Solving Structural Conflicts. *IEEE Transactions On Knowledge and Data Engineering*, 6(2):258–274, 1994.

[ST99] R. Shamir and D. Tsur. Faster Subtree Isomorphism. *Journal of Algorithms*, 33(2):267–280, November 1999.

[TAB+97] A. Tomasic, R. Amouroux, P. Bonnet, O. Kapitskaia, H. Naacke, and L. Raschid. The Distributed Information Search Component (Disco) and the World Wide Web. In *SIGMOD Record*, volume 26 of *2*, pages 546–548, May 13–15 1997.

[TH04] I. Tatarinov and A. Y. Halevy. Efficient Query Reformulation in Peer-Data Management Systems. In *ACM SIGMOD Conference*, 2004.

[TIM+03] I. Tatarinov, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza peer data management project. *SIGMOD Record*, 32(3), 2003.

[TRS97] M. Tork-Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 266–275, 1997.

[Ull97] J. Ullman. Information Integration Using Logical Views. In *Proceedings of the International Conference in Database Theory (ICDT)*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40, Delphi, Greece, 8–10 January 1997. Springer.

[VCC00] Y. Velegrakis, V. Christophides, and P. Constantopoulos. On Z39.50 Wrapping and Description Logics. *International Journal of Digital Libraries*, 3(3):208–220, 2000.

[VMM05] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Managing and Querying Schema Transformations. In *Proceedings of International Conference on Data Engineering (ICDE)*, 2005.

[VMP03a] Y. Velegrakis, R. J. Miller, and L. Popa. Adapting mappings in frequently changing environments. Technical Report CSRG-468, Univ. of Toronto, Dep. of Comp. Sc., February 2003. ftp://ftp.cs.toronto.edu/cs/ftp/pub/reports/csri/468.

[VMP03b] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 584–595, 2003.

[VMP04] Y. Velegrakis, R. J. Miller, and L. Popa. On Preserving Mapping Consistency under Schema Changes. *International Journal on Very Large Data Bases*, 13(3):274–293, 2004.

[VMPM04] Y. Velegrakis, R. J. Miller, L. Popa, and J. Mylopoulos. ToMAS: A System for Adapting Mappings while Schemas Evolve, (System Demonstration). In *Proceedings of International Conference on Data Engineering (ICDE)*, 2004.

[W3C01] W3C. XML Schema Part 0: Primer. http://www.w3.org/TR/xmlschema-0/, May 2001. W3C Recommendation.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 25–30, Baltimore, Maryland, 1995.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[Win02] A. Winter. GXL - Overview and Current Status. In *International Workshop on Graph-Based Tools (GraBaTs)*, 2002.

[WM90] Y. R. Wang and S. E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 519–538, 1990.

[YMHF01] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *ACM SIGMOD Conference*, pages 485–496, 2001.

[YP04] C. Yu and L. Popa. Constraint-Based XML Query Rewriting For Data Integration. In *ACM SIGMOD Conference*, 2004.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *ACM SIGMOD Conference*, pages 316–327, 1995.

Index