12-3-2019

Local Radiance

Scott Peter Britell
Portland State University

Local Radiance

by

Scott Peter Britell

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Lois Delcambre, Chair
David Maier
Paolo Atzeni
Gerald Recktenwald

Portland State University
2019

ABSTRACT

Recent years have seen a proliferation of web applications based on content management systems (CMS). Using a CMS, non-technical content authors are able to define custom content types to support their needs. These content type names and the attribute names in each content type are typically domain-specific and meaningful to the content authors. The ability of a CMS to support a multitude of content types allows for endless creation and customization but also leads to a large amount of heterogeneity within a single application. While this meaningful heterogeneity is beneficial, it introduces the problem of how to write reusable functionality (e.g., general purpose widgets) that can work across all the different types.

Traditional information integration can solve the problem of schema heterogeneity by defining a single global schema that captures the shared semantics of the heterogeneous (local) schemas. Functionality and queries can then be written against the global schema and return data from local sources in the form of the global schema, but the meaningful local semantics (such as type and attribute names) are not returned. Mappings are also complex and require skilled developers to create.

Here we propose a system that we call *local radiance* (LR) that captures both global shared semantics as well as local, beneficial heterogeneity. We provide a formal definition of our system that includes domain structures—small, global schema fragments that represent shared domain-specific semantics—and canonical

structures—domain-independent global schema fragments used to build generic global widgets. We define mappings between local, domain, and canonical levels. Our query language extends the relational algebra to support queries that radiate local semantics to the domain and canonical levels as well as inserting and updating heterogeneous local data from generic global widgets. We characterize the expressive power of our mapping language and show how it can be used to perform complex data and metadata transformations. Through a user study, we evaluate the ability of non-technical users to perform mapping tasks and find that it is both understandable and usable. We report on the ongoing development (in CMSs and a relational database) of LR systems, demonstrate how widgets can be built using local radiance, and show how LR is being used in a number of online public educational repositories.

DEDICATION

To Beth and Sam, whose support and encouragement give me the strength to accomplish the hardest challenges.

ACKNOWLEDGMENTS

This work would not be possible without the guidance and patience of my advisor, Lois Delcambre, to whom I owe many thanks. She has helped me grow into someone who sees opportunities instead of problems. Through peaceful disagreement and violent agreement, she has helped me become a better writer, researcher, and mentor to others.

Thanks also to Dave Maier, my co-advisor, who has patiently listened and supported my work and helped with the smallest details.

I give my sincere thanks to Paolo Atzeni for his support throughout the years not only as a member of my committee, but also as a co-author on numerous papers. Paolo has guided us through the various iterations of formalism presented in this work. He has patiently read more versions of this work than one might reasonably expect.

Thanks to Gerald Recktenwald for joining my committee midway through this process, supporting my work, and for being deeply interested in more things than one generally has time for.

I would like to thank Bernhard Thalheim for shepherding us through HERM and helping us better understand the nature of our work.

This work began with the creation of a website to host Randy Steele's and Don Domes's K-12 robotics curriculum. This website has been the testing ground for the work described in this thesis. I owe a great debt of gratitude to Randy for allowing us to experiment and for his dedication and support throughout the

years.

Without the hard work of many students there would be no local radiance systems. Thanks to the ASE apprentices Ananth Mohan, Stanley Cen, Sarah Lemieux, and Austin Chang who were able to take my ideas and make them reality. Thanks to Jason Owens who taught me the right way to do many things. The Drupal query and mapping interfaces exist due to Andrew Hobbs and Cameron Hobbs. Thanks to Laura DeWitt for helping build the mapping specifications and widget development. Special thanks to Iman Bilal for her help in getting IRB approval and setting up user studies.

I thank all of the faculty and students that were part of the Ensemble team. I especially thank our collaborators Ed Fox, Lillian Cassel, Dan Garcia, Richard Furuta, Peter Brusilovsky, Greg Hislop, Frank Shipman, Weiguo Fan, and Bob Siegfried for their advice and encouragement. I also owe thanks to Monika Akbar and Yilin Chen for teaching me all the gory details under the hood of the Ensemble site.

To Abdusullam Alawini, Dave Archer, Hisham Benotman, Rafael de Jesus Fernández Moctezuma, Michael Grossniklaus, Veronika Megler, Christoph Schütz, Jeremy Steinhauer, Kristin Tufte, and all the members of Datalab, I am grateful for many years of discussion and friendship. I particularly thank Len Shapiro for starting me on this journey.

This work was supported, in part, by National Science Foundation grants 0840668 and 1250340.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

There are currently over 1.5 billion websites on the World Wide Web [42]. Of those sites, almost 57% (or roughly 850 million) use a content management system (CMS) [83] that allows non-technical end-users to create complexly structured data that is shown and manipulated in websites and web applications. Using a CMS, non-technical users are able to define custom content types to support their needs. For example, in a repository of educational materials, a teacher may decide to create content for a course, its units, lessons, assessments, and instructional materials. To do so, the teacher creates content types for each part of the course with a field for the title of the content and separate fields for different data for each of these content types. For example, for a course there may be a field for the appropriate grade level of the material, for a unit there may be a field for subject area, and for an instructional material there may be a field for the presentation type of the content. Once the content types are created and populated the data is immediately available in the CMS and presented as webpages where the attribute names of each content type are displayed along with the data itself. These features of the CMS allow each user to create a set of content types with a conceptual model that suits their needs.

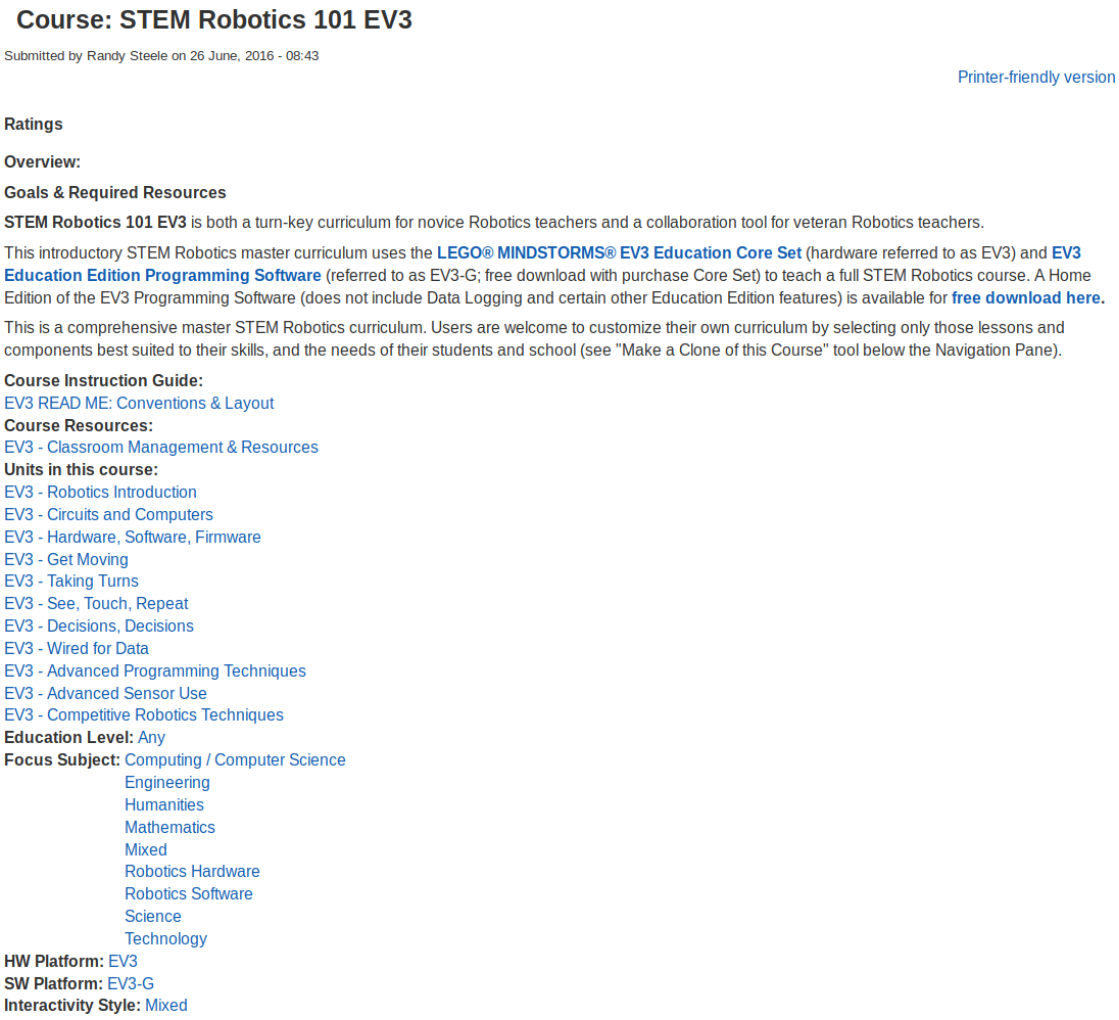
Figure 1.1 shows a course webpage in the STEMRobotics educational repository[1] created by a middle school teacher using the Drupal CMS [33]. The course

[1]https://stemrobotics.cs.pdx.edu, accessed 11-10-2019

has fields for "Overview", "Education Level", "Focus Subject", "HW Platform", "SW Platform", and "Interactivity Style". Some fields may hold directly entered text like the "Overview" field, while others may hold a value chosen from a controlled vocabulary such as "Education Level", "Focus Subject", "HW Platform", "SW Platform", and "Interactivity Style". The course webpage also has links to an instruction guide, course resources, and units. This course is designed to provide teachers with all the materials necessary to teach robotics to middle school students and comprises units that have lessons which, in turn, have different types of assessments and instructional materials.

A high school teacher using the same STEMRobotics repository created the challenge-based course shown in Figure 1.2. This course also has a text entry field for "Overview" and fields for "Education Level", "HW Platform", and "SW Platform". This course is self-directed, where students are expected to figure out how to finish each challenge and pass the course by demonstrating the successful completion of all challenges. In both figures, the title is shown at the top of the webpage, just after the content-type name.

Both teachers were able to easily create course structures that suited their own needs even though neither one is a web developer. The ability of a CMS to support a multitude of content types allows for endless creation and customization, but also can lead to a large amount of heterogeneity within a single application. For example, if ten teachers each want a custom course structure, they can each build one. We see this heterogeneity as important and useful. For example, challenges, quizzes, and oral exams are all forms of assessment in the educational domain. However, a challenge may have associated instructional materials and oral exams may have time limits and committees. This additional information is important; if we were to impose a fixed assessment structure we could lose this information. We call this type of heterogeneity with semantically meaningful names, *beneficial heterogeneity*.



Course: STEM Robotics 101 EV3

Submitted by Randy Steele on 26 June, 2016 - 08:43

Printer-friendly version

Ratings

Overview:

Goals & Required Resources

STEM Robotics 101 EV3 is both a turn-key curriculum for novice Robotics teachers and a collaboration tool for veteran Robotics teachers.

This introductory STEM Robotics master curriculum uses the **LEGO® MINDSTORMS® EV3 Education Core Set** (hardware referred to as EV3) and **EV3 Education Edition Programming Software** (referred to as EV3-G; free download with purchase Core Set) to teach a full STEM Robotics course. A Home Edition of the EV3 Programming Software (does not include Data Logging and certain other Education Edition features) is available for **free download here.**

This is a comprehensive master STEM Robotics curriculum. Users are welcome to customize their own curriculum by selecting only those lessons and components best suited to their skills, and the needs of their students and school (see "Make a Clone of this Course" tool below the Navigation Pane).

Course Instruction Guide:
EV3 READ ME: Conventions & Layout
Course Resources:
EV3 - Classroom Management & Resources
Units in this course:
EV3 - Robotics Introduction
EV3 - Circuits and Computers
EV3 - Hardware, Software, Firmware
EV3 - Get Moving
EV3 - Taking Turns
EV3 - See, Touch, Repeat
EV3 - Decisions, Decisions
EV3 - Wired for Data
EV3 - Advanced Programming Techniques
EV3 - Advanced Sensor Use
EV3 - Competitive Robotics Techniques
Education Level: Any
Focus Subject: Computing / Computer Science
Engineering
Humanities
Mathematics
Mixed
Robotics Hardware
Robotics Software
Science
Technology
HW Platform: EV3
SW Platform: EV3-G
Interactivity Style: Mixed

Figure 1.1: A course webpage produced by the Drupal CMS showing attribute names like “Overview”, “Education Level”, “Focus Subject”, “HW Platform”, “SW Platform”, and “Interactivity Style” with their associated data. Relationships to other content types are also shown with links to a “Course Instruction Guide” and “Units in this course”.

Challenge-based Course: Don Domes Robotics Course

Submitted by Don Domes on 22 July, 2011 - 14:58

Printer-friendly version

Overview:

Robotics 1 is a level one class at Hillsboro High School. Any student can enter the class with no prerequisites. The class is designed to be a foundation and exploration class leading to advanced studies in our technical and engineering courses. Typical follow on courses are Electronics, Auto, Computer Science, Drafting and Design, and Physics. The typical class is composed primarily of freshmen and sophomores. Generally at least 35% of the class is nontraditional populations consisting of non-Caucasian students and girls. Approximately 40% of our school population is Latino. We have several of our resources translated into Spanish.

In Robotics 1 the LEGO Mindstorms kits (with the yellow RCX computer brick) are the learning platform we use. We obtained over 40 Mindstorms kits in the early years of the program and have a proven curriculum that attracts students to the class.
Class sizes of 28 to 35 are typical with two to four sections of the course subscribed every year for the last five years. The course sequence is design to help high school students understand basic programming, mechanical, and science/math concepts. The relation these concepts have to physical science and computer science are stressed to help students make connections to other areas of study.

Challenges:
Incline Plane Challenge
Can Do Challenge
Repetitive L – Accuracy Challenge
Three Squares - Accuracy Challenge
Robot Control
Line Follower
Education Level:
High School
HW Platform:
NXT
SW Platform:
NXT-G

Figure 1.2: A challenge-based course webpage produced by the Drupal CMS showing the attribute names "Overview", "Education Level", "HW Platform", and "SW Platform" with their associated data. The relationship between the course and its instructional materials is shown with the links to "Challenges".

For some, just creating a website with their data is enough, but for most website creators there is a need or desire to add further functionality to their website. For example, a teacher may want to show the hierarchical structure of a course in a tree-based browsing widget; or, for a particular term that a course is offered a teacher may wish to have the dates that lessons are taught appear in a calendar widget; or, a course that is taught in various locations may benefit from a map widget. A user of the site may also wish to search for all assessments, regardless of the type of course they appear with and regardless of the local fields they include.

We would like to distinguish between the types of users of a CMS. The non-technical content creators described above we call *content authors*. Content authors have a deep understanding of their content domains and typically can create their own types and populate them with data. But often they do not have the technical expertise to create or install widgets. Users that create, install, and manage widgets we call *widget developers*.

Currently, to enable a widget in a CMS a user must use the predefined content types associated with the widget. For example, calendar widgets will typically use a "Calendar Entry" type and a map will use a "Location" type. If the types are not used (or even not fully populated in many cases) the widgets will fail to work. These fixed types can also cause beneficial heterogeneity to be lost by forcing content authors to conform to the widget types. The functionality of the widgets is also predefined and any content author wanting to have the widget work slightly differently for their content will need to work with a developer to reconfigure the widget in the CMS or possibly redevelop the widget altogether.

The current situation raises the question: how can we enable the use of generic widgets while maintaining beneficial heterogeneity? If we look to the field of information integration [48], we can see the conceptual model created by our content authors as analogous to local schemas and the model of the widgets as analogous

to global schemas. The global schema captures the shared semantics of the heterogeneous local schemas and users (with the appropriate technical skills) create mappings between the local and global schemas. Functionality and queries work against the global schema and an information integration system uses the mappings to retrieve data from the local schemas.

This approach exists in the web most notably through the semantic web [6]. Technologies such as XML [88], GRDDL [38], RDF [71], and OWL [84] allow the semantics of webpages to be formally defined and mapped to global ontologies (schemas). Functionality and queries working against a global schema in the web often take the form of widgets that can be placed in a web page.

However, there are a number of limitations to this type of (traditional) information integration. Widgets written against the global schema are only able to access data in the form of the global schema. Beneficial heterogeneity is not accessible through the widgets. The local schema names are not available from the global schema which prevents the website users from seeing the content type names and field names (from the local schema) that were originally chosen by the content author. These names are likely meaningful and useful to website users because they are likely familiar with the domain.

Global schemas are usually also large and fixed which can fail to capture smaller independent shared semantics. For example, there may be prerequisite relationships between courses at different granularities, such as intercourse prerequisite relationships for some courses and intra-course relationships (e.g., lesson sequencing) for others. In this case, a fixed global schema containing both course and lesson structures would then need multiple different prerequisite relationships even though they are conceptually the same relationship (e.g., from the point of view of a widget displaying such dependencies). Large global schemas are also typically time-consuming and difficult to build, requiring the agreement of the local schema participants. Within the semantic web, the goal has been to define unified or deep

ontologies, but the usage of these ontologies has been slow to materialize. In 2010, Anderson and Lee wrote about the relatively slow uptake of the semantic web after 10 years of use and asked experts whether the original semantic web vision is likely to be achieved by the year 2020 [2]. In response, the experts felt the most obvious explanations for the slow uptake were that it requires too much work and that the benefits are not clear enough to warrant a wholesale migration.

Another limitation to this type of integration is the work involved in creating mappings between user content and global schema. Mappings often need to be defined in formats and query languages not accessible to non-technical users. Global schemas often also need be fully mapped before they are functionally useful.

If fully mapping global schemas is difficult, can we achieve this type of functionality using schema fragments or small patterns? Within the semantic web, this concept has been realized through the use of technologies such as RDFa [70] and Microformats [55], which allow content creators to embed RDF relationships or schema fragments directly within a webpage. Embedding known schema elements from sources such as Schema.org [74] allows content creators to quickly add semantics to their content and to take advantage of search engine and other functionality that exploits these schema elements. A similar web mashup tool, Paggr [60], uses the collection of RDF data known on the web to build reusable and extensible web page widgets written using SPARQL [77] as a common interface to the web of data. That approach allows widgets to show information contained within the web of data, but can only show data that has been directly mapped in RDF. These systems also require users to understand RDF and create RDF mappings within their sites leading to the question: how can we provide a way to map information that does not require complex technical knowledge? In information integration, the Clio project [57] showed that schema mapping could be achieved using graphical interfaces where users simply drew lines between local and global schemas. Can we also bring this style of mapping to content authors in a CMS?

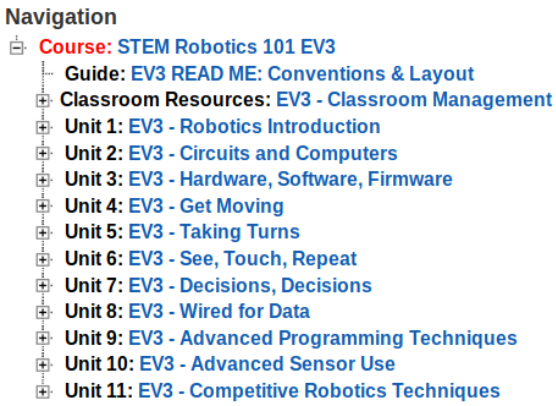


Figure 1.3: A tree-based navigation widget for the course in Figure 1.1.

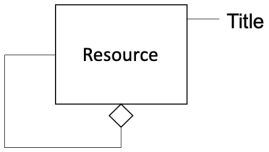


Figure 1.4: A schema fragment for a parent-part hierarchy.

It is often the case that a widget uses a very small schema. For example, a navigation widget, shown in Figure 1.3, for the course-unit structure from the website in Figure 1.1 is built with a generic data model such as that shown in Figure 1.4. The widget only needs to know the title of a resource and the recursive relationship with the children of the resource. The widget developer does not need to know that a resource and its children have different types and fields.

Another widget, for searching, may use a model of a generic entity with a title and a number of attributes with which it indexes and creates facets with a model like that shown in Figure 1.5. A developer may understand what the various attributes are for, but it is unlikely that the content author will understand this model. What happens when the model of a widget is so generic that a content author may not understand or know how to use it?

One solution to this problem is to introduce domain information in to the widget schemas. Solutions such as topic maps [81] and structured maps [31] provide

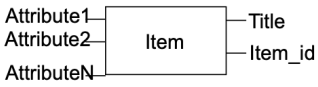


Figure 1.5: A generic widget model.

domain meta-models that encompass common semantics and structures within a domain, and are generally designed to provide better browsing and searching of source objects in that domain. The information in the map may not exist explicitly in the source objects. Superimposed information [52] systems have provided methods for easily mapping base information models to these types of meta-models through the use of marks, where a mark is an address for a segment of base information. Using this type of system, a user can easily highlight and extract the information of interest within their source data to enable functionality at a global level, without having to modify their existing data. These systems work in large part because the meta-models are domain-specific and understandable by their information creating users (content authors) and end users. How can we leverage the power of domain models and superimposed information to enable our content authors to more easily benefit from the power generic schema of widgets, while not forcing authors to modify their existing data?

1.1 OUR SOLUTION

While all of these techniques provide some part of a solution to providing generic functionality to end-user-developed sites, none of them will directly allow us to bring beneficial heterogeneity to the global level. Take the navigation widget in Figure 1.3, for example. The local type names (e.g., “Course”, “Unit”, and “Guide”) are meaningful and let the website user know that there is a difference between resources that appear at the same level of the hierarchy. We believe local names are important, especially when they differ from the base semantic concept

that they specialize (otherwise why were they chosen by the content authors?).

This thesis presents our system, which we call *Local Radiance* (LR), where local radiance refers to the capability to let local schema (and data) "shine" through to the global level. It is built upon the principle tenets that local content authors understand their own data and have chosen names that are important to them, understand their domain, and can create mappings between their data and domain concepts. Our aim is to answer the following research questions.

How can we enable information integration that retains local beneficial heterogeneity? Chapters 2 and 3 present our LR system. We define *canonical structures* that are small global schema fragments that widgets are written against. We define *domain structures* that are small domain-specific schema fragments that capture shared semantics within a domain. We show how users create mappings between canonical and domain levels as well as between domain and local levels. We then define an extended relational query language that can be used from the canonical or domain levels to write queries that can present data mapped to the canonical or domain levels while radiating local schema names.

How can we enable non-technical end-user schema mapping and information integration? Traditional information integration and schema mapping is difficult; it often requires in-depth knowledge of both local and global schemas as well as complex technical knowledge to create mappings. We limit the type of mappings that can be created in our system to a simple form such that it will be understandable to non-technical users. By making schema mapping as easy as possible and limiting mappings to be between just one entity at the domain level just one entity at the local level, we believe users will be able to understand the process and complete it correctly. We offer widgets as an incentive for content authors to create mappings. In Chapter 2, we present the results of a user study to test this hypothesis.

How can we build generic widgets that capture beneficial heterogeneity? Our base query language, presented in Chapter 3, provides operators for integrating and accessing local information from the domain and canonical level as well as operators that radiate local type and attribute names to the domain and canonical levels. We show how queries using our operators return all local type and attribute names, which can be used to build widgets for searching, browsing, and navigation. While the base LR system is defined using a nested relational model, we show how the system can act as a standard relational system minimizing the impact on widget developers such that they need only learn to use a few new operators instead of learning to use an entirely new model and query language.

Can we leverage local radiance to create generic local data-creation and data-manipulation widgets? Our base LR system provides read-only querying capabilities similar to views with the addition of access to local beneficial heterogeneity. While this capability is important and enables a wide range of useful widgets (e.g., searching, browsing, and navigation), is it also possible to build widgets that create and update local data when there is beneficial heterogeneity? And, can we do so when local schemas are mapped only to small schema fragments and maybe only partially mapped? Chapter 4 presents extensions to our base query language that enables widgets to access all local information even when there is only a partial mapping. We define operators for both insert and update of local data that can be used generically at the canonical and domain levels.

Can we empower end-users to perform complex data transformation tasks? In Chapter 5 we show that our system can be extended to perform data-to-metadata and metadata-to-data transformations. We extend our mapping definitions and show how widgets can be built that allow end-users to perform complex data transformation tasks.

What is the best way to formalize and implement an LR system? Chapters 2 and 3 present the formalism of our LR system. In Chapter 6 we

present the evolution of local radiance. We show how each step in the evolution furthered the development of LR as well as its formal description.

1.2 STRUCTURE OF THIS DOCUMENT

Chapter 2 presents a formal definition of the conceptual model of the LR system. We introduce our common global patterns: domain and canonical structures and explain why both levels are needed. We define our mapping model and we present the results of our user study.

Chapter 3, presents our base query language. We define the *apply* and *type* operators that integrate local data and radiate local schema names respectively. These operators allow widgets to use polymorphic queries. We also show how to optimize queries using these operators.

Chapter 4 extends our query language with operators for insert and update of local data from queries against global schema. Generic widgets use these operators for local data manipulation.

Chapter 5 demonstrates how our system can be extended to perform data-metadata transformations used for traditional transformation tasks such as pivot and unpivot. We demonstrate these transformations in a digital library widget that creates complex, faceted searching.

Chapter 6 describes the different iterations of the implemented LR system and their associated formal definitions. We present the ways in which generic widgets have been written in the system. We show how widgets can be modified through the use of *mapping specifications* to enable content authors to modify widget functionality through mappings.

We conclude in Chapter 7 with a presentation of future work possibilities and a description of publications to date.

Chapter 2

STRUCTURES AND MAPPINGS

Technologies such as web development frameworks have democratized the creation of complex systems by allowing non-technical (non-developer) content authors to define their own content types and create complex data models (i.e., conceptual models), while abstracting away the complexities of database and application creation. As a consequence, content authors who are experts in their own data can choose schema names that are meaningful. We call content-author-created schemas *local schemas* and their associated data *local databases*.

Modern web frameworks also allow developers to create widgets that plug into any site built upon that framework. Widgets typically add features beyond the data presentation of the site, such as browsing, searching, maps, calendars, etc. These widgets use a schema (conceptual model) of the developer's choosing that is typically related to the functionality of the widget.

Traditionally, in order for a widget to work on a user's web site, there are two choices. A developer may rewrite the same widget multiple times for the different conceptual models of the end systems. For example, in the case of a calendar widget, the developer could modify the widget to work with each different event type. Or, the end systems could rename their schema elements to match the model of the widget; in the case of the calendar widget, each end-user would have to use the event type defined by the widget. The latter is the common case in use today by most web development frameworks.

We present a different solution to this problem. We want to entice users to perform traditional schema-mapping and data-integration tasks by providing generic

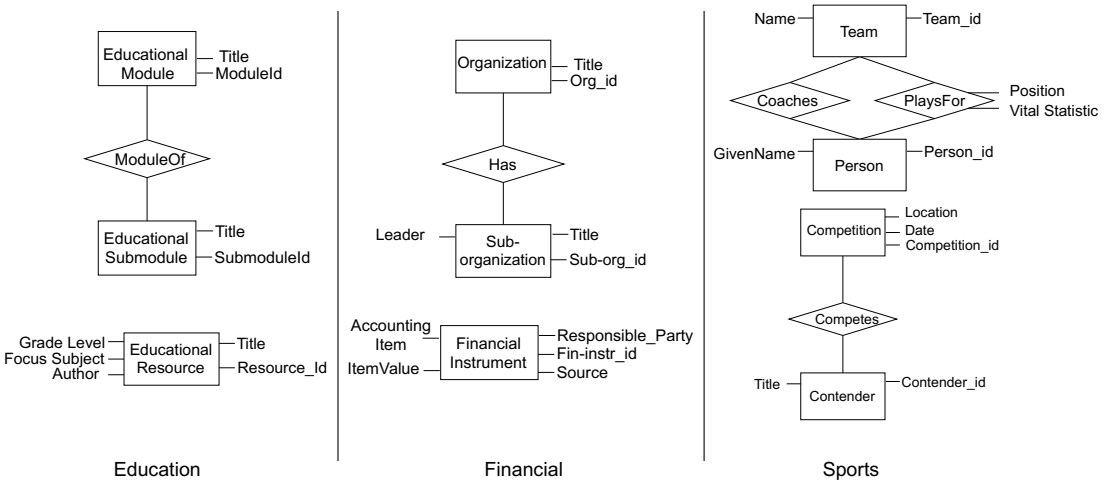


Figure 2.1: Examples of domain structures from the educational (left), financial (middle), and sports (right) domains.

widgets that can be added to existing websites. We begin by introducing intermediary conceptual models, that we call *domain structures*, that represent notable, commonly occurring patterns within a domain. For example, in Figure 2.1 we see domain structures for the educational, financial, and sports domains. The educational domain structure (left side of the figure) shows a relationship pattern between educational modules and submodules as well as the pattern for an educational resource that contains common attributes of educational resources (i.e., "Grade Level" and "Focus Subject"). The patterns within the financial domain structure (middle of the figure) appear structurally similar to those of the educational patterns but are semantically quite different (an organizational structure of a business versus that of courses and materials, and financial instruments versus educational resources). The sports domain structures (right side of the figure) is structurally different from the previous two with a relationship describing contenders in competitions (such as teams playing a match or boxers in a fight) and the relationships and entities that define a team structure (teams that have players and coaches).

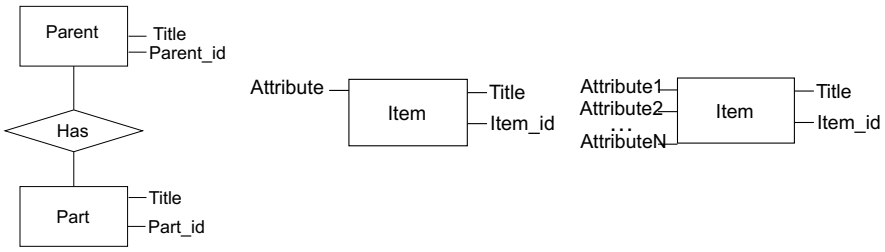


Figure 2.2: Three examples of canonical structures.

We then define models for the widgets, that we call *canonical structures*. A canonical structure is usually rather simple, essentially a "data pattern", on top of which widget code is implemented. A canonical structure often involves a single entity (such as those shown in the middle and right in Figure 2.2), to be used by widgets that manage (search, analyze, update, etc.) objects (such as those described by either of the "Item" entities) of a given data type (phone books, recent messages, calendars, etc.). And widgets will often manage collections or hierarchies of things which can be captured with the "Parent"–"Part" relationship on the left side of the figure. A widget can often be reused across a variety of domains and therefore benefits from a generic structure that can be instantiated many times.

We define mappings (such as those used in traditional information integration [48] and schema mapping [57]) between the different levels (local schemas-domain structures and domain structures-canonical structures). Content authors can create local schemas with meaningful names and widget developers can create generic widgets with canonical structures. And, we allow the generic widgets to show the local schema names using what we call *local radiance*.

Our system can be used by people in three main roles. We consider the content author to be a domain expert since someone creating an application for their data should understand their domain. The content author is responsible for deciding the local schema and data which will be used in the system. This person will

enable widgets by creating mappings between the local schemas and the domain structures.

As described in the previous chapter, we call the developer responsible for creating generic widgets the widget developer. This person writes widget code that interacts with generic schemas, the canonical structures, that produce information that can be displayed on a webpage or used elsewhere in a web framework.

We define domain structures that are (typically small) schemas with names that are understandable to a content author representing common semantics within a domain of interest. We then add a third role to the two traditional roles: the *domain developer*, whose responsibility it is to create mappings between the schema of the domain structures and the generic schemas (the canonical structures) of the widgets. The domain developer usually has some (possibly in-depth) knowledge of the domain but their main responsibility is more likely IT-based (database, web, or application development) rather than content creation. Domain structures will typically be defined by a domain developer. This person may work with the content author to create the website and may work with widget developers to allow the generic widgets to be used in specific application areas.

In this chapter we will show how schema is defined for use by each of these types of users and how mappings can be created between the schema levels. We present the conceptual model of our system and its formalism and we make the following contributions:

- We define *local databases* that are typically end-user- (content author-) created schemas and databases.
- We define *domain structures* that represent common patterns within a specific domain.
- We define *canonical structures* that represent generic structures used across many domains.

- We define our mapping system that allows mappings between local databases and domain structures; and, between domain structures and canonical structures, thereby enabling generic widgets to work with heterogeneous local databases.

- We define the scope of mappings with our system and compare how our mappings compare to traditional tuple-generating dependencies, a common mechanism for database information integration.

- We evaluate the use of our mapping system by non-technical and technical users through a user study.

2.1 STRUCTURES AND MAPPINGS

We show representations of our structures using the Entity-Relationship (ER) [25] model to illustrate schema and mappings. In implementations, local databases, domain structures, and canonical structures use the relational model to present data to end-users. We use the nested relational model [73] to represent local, domain, and canonical schemata as well as mappings (equivalent to a system catalog). We show examples in the ER, relational, and nested relational models.

We present local, domain, and canonical schemata and mappings formally through the use of a running example. Figure 2.3 shows three local schemas within an educational domain. The leftmost structure represents a traditional course where the "Course" is made up of one or more "Units" and each "Unit" consists of one or more "Lessons". A "Lesson" may have associated instructional materials. A standards-based course, as shown in the right side of Figure 2.3 has a structure that is similar to a lesson in the traditional course but with different names for relationships than those used in lessons. The middle structure in Figure 2.3 shows a "Tutorial Course" which consists of "Tutorial Units" where each unit may consist of any number of instructional materials that are tutorials.

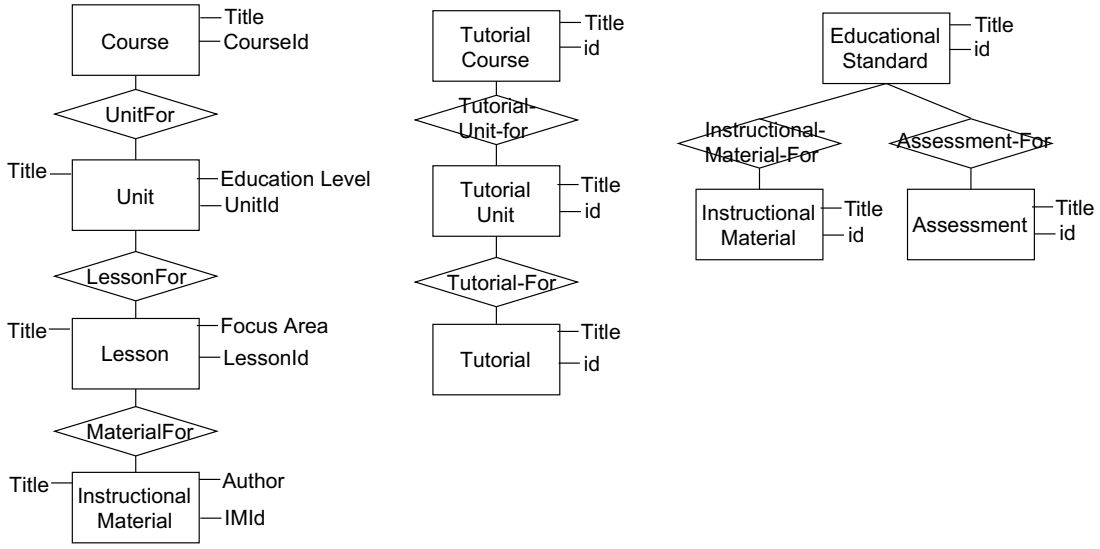


Figure 2.3: Three local schemas within the educational domain. These schemas have been simplified for clarity in the examples throughout this chapter.

We describe how the local database schemas are represented in our system much as relational databases store system catalogs (here, we use a nested relational model); this information can be used later by widgets to extract local type names generically (based on the mappings that have been created).

Local DBs:

$ldb(\underline{id}, lrs(name, key, attrs(name)))$

A local database is defined by a tuple in the nested relation ldb. Each database is defined by an identifier id and a nested relation of the local relation names in the database. Each nested local relation tuple contains the name of a local relation, the name of the key attribute of the local relation, and a nested relation of the attribute names in that local relation. Our system has been designed within the world of web information systems and databases and as such each entity (i.e., each relation in the local schema or, equivalently, each content type) will have a unique identifier (commonly a URL). Working in the environment of the web where all data is converted into strings for display on web pages, we ignore local datatypes

and treat all canonical and domain attributes as strings.

Result Set 2.1 shows the definition of the local schema on the left of Figure 2.3[1]. The "Course", "Unit", "Lesson", and "Instructional Material" entities have been directly translated into local relations and the three relationships are also represented by relations and have an associated identifier for each participant.

Result Set 2.1

```
id             |                lrs
---------------+-------------------------------------------------------------------
StandardCourse| {(Course,CourseId,{CourseId,CourseTitle}),
               |  (Unit, UnitId, {UnitId, UnitTitle, EducationLevel}),
               |  (Lesson, LessonId, {LessonId, LessonTitle, FocusArea}),
               |  (InstructionalMaterial, IMId, {IMId, IMTitle, MaterialType}),
               |  (UnitFor, UnitForId, {UnitForId, CourseId, UnitId}),
               |  (LessonFor, LessonForId, {LessonForId, UnitId, LessonId}),
               |  (MaterialFor, MaterialForId, {MaterialForId, LessonId, IMId})}
```

To help clarify our definition of the the local database we show the standard relational schema of the "StandardCourse" local database below (key attributes are underlined).

Course($\underline{CourseId}, CourseTitle$)

Unit($\underline{UnitId}, UnitTitle, EducationLevel$)

Lesson($\underline{LessonId}, LessonTitle, FocusArea$)

InstructionalMaterial($\underline{IMId}, IMTitle, MaterialType$)

UnitFor($\underline{UnitForId}, CourseId, UnitId$)

LessonFor($\underline{LessonForId}, UnitId, LessonId$)

MaterialFor($\underline{MaterialForId}, LessonId, IMId$)

We define domain structures in a similar fashion to the local databases.

Domain Structures:

$ds(\underline{id}, drs(name, key, attrs(name)))$

[1]Note, throughout this chapter we will use result sets to show the nested relational examples. The result sets are available from a PostgreSQL implementation of our formalism and will be explained in further detail in the next chapter.

A domain structure is defined by a tuple in the nested relation ds. Each domain structure is defined by an identifier id and a nested relation of the domain relation names in the structure. Each nested domain relation tuple contains the name of a domain relation, the name of a key attribute, and a nested relation of the attribute names in that domain relation. The key attribute of the domain structure is not a traditional relational key, but rather, is generated in the mapping process (described below), and uniquely identifies the local database, local relation, and local tuple from which the domain relation tuple was derived. This key may or may not uniquely identify the domain relation tuple depending on the mapping that was used to create the tuple. This nested relation, ds, stores the schema of all domain structures. For example, the educational domain structure from Figure 2.1 is represented below in Result Set 2.2.

Result Set 2.2

```
id             |                drs
---------------+-------------------------------------------------------------------
EducationalDS  | {(EducationalResource,ResourceId,{ResourceId,Title,GradeLevel,
               |                                    FocusSubject,Author}),
               |  (EducationalModule, ModuleId,{ModuleId,Title}),
               |  (EducationalSubmodule, SubmoduleId,{SubmoduleId,Title}),
               |  (ModuleOf, ModuleOfId, {ModuleOfId, ModuleId, SubmoduleId})}
```

The domain structure contains the following four relations.

EducationalResource($\underline{ResourceId}, Title, GradeLevel, FocusSubject, Author$)

EducationalModule($\underline{ModuleId}, Title$)

EducationalSubmodule($\underline{SubmoduleId}, Title$)

ModuleOf($\underline{ModuleOfId}, ModuleId, SubmoduleId$)

Mappings can then be created between the local databases and the domain structures (typically by the local database creator or content author). Figure 2.4 shows two mappings that have been created between the "ModuleOf" domain relation (left side of Figure 2.1) and the "StandardCourse" local database (left side of Figure 2.3). Each set of colored lines represents a single mapping consisting of a

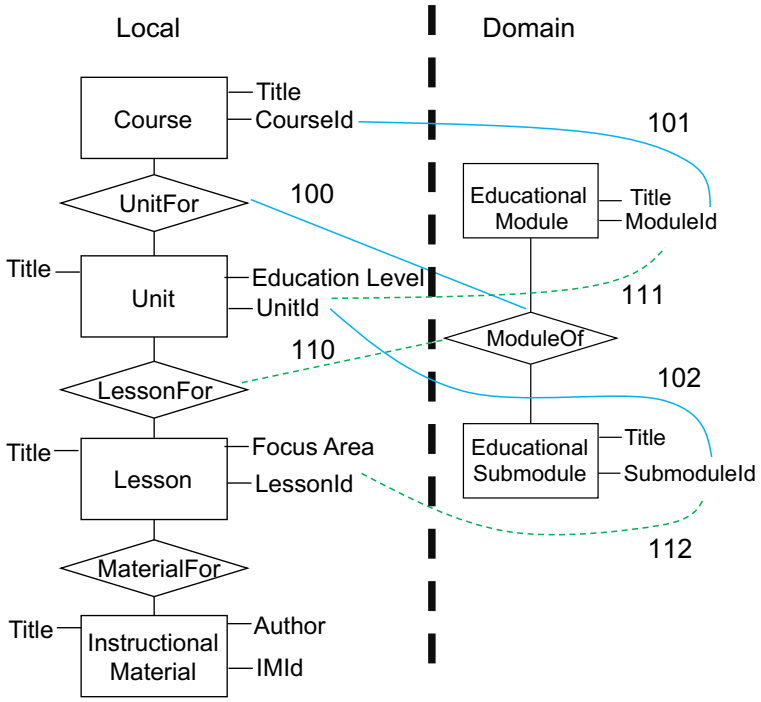


Figure 2.4: Mappings are shown between the "ModuleOf" domain structure and the "UnitFor" local relationship (blue, solid lines) and the "LessonFor" local relationship (green, dashed lines). Correspondence ids are added to show the correspondences listed in Result Set 2.3. Correspondences ids are auto-generated by the system and not visible to end-users.

number of individual correspondences. As shown in the figure, each correspondence has its own id, shown as a number with the line of the correspondences and each mapping will have its own id. Mapping and correspondences ids will not typically be user-generated, but rather system-generated as users create mappings through simple-to-use interfaces.

Local DB - Domain Structure Mappings:

$ds\_ldb\_m(\underline{id}, ldbid, dsid, dr\_lr\_ms(id, lr, dr, p, corrs(id, la, da)))$

The set of all local database to domain structure mappings is defined in the ds_ldb_m nested relation. A mapping is defined as one tuple in the relation with an identifier id, the local database in the mapping $ldbid$, the domain structure in

the mapping $dsid$, and a nested relation of the relations mapped dr_lr_ms. Each mapping tuple (between a domain relation and a local relation) in dr_lr_ms consists of an identifier id, the local relation lr, the domain relation dr, a predicate p, and a nested relation of the correspondences between local and domain attributes. Each correspondence consists of an identifier id, the local attribute name la and the domain attribute name da. The predicate in a mapping will be used in conditional mappings and described in more detail in Section 2.2.4. The conditions for a well-formed mapping are presented in Section 2.2.1. The mappings from Figure 2.4 are shown below in Result Set 2.3.

Result Set 2.3

id	ldbid	dsid	dr_lr_ms
1	StandardCourse	EducationalDS	{(10,UnitFor,ModuleOf,TRUE,{(100,UnitForId,ModuleOfId), (101,CourseId, ModuleId), (102,UnitId, SubmoduleId)}), (11,LessonFor,ModuleOf,TRUE,{(110,LessonForId,ModuleOfId), (111,UnitId, ModuleId), (112,LessonId, SubmoduleId)})}

Canonical structures are defined similarly to local databases and domain structures.

Canonical Structures:

$cs(\underline{id}, crs(name, key, attrs(name)))$

Each canonical structure is defined by a tuple in the nested relation cs. Each canonical structure is defined by an identifier id and a nested relation of the canonical relation names in the structure. Each nested canonical relation tuple contains the name of a canonical relation, the name of a key attribute, and a nested relation of the attribute names in that canonical relation. The key attribute value of the canonical relation will come directly from the key attribute value from each domain relation that is mapped to the canonical relation. Hence, the key may or may not uniquely identify tuples in the canonical relation depending on the local-relation-to-domain-relation mapping that created the key value. This nested relation contains the schema for all canonical structures. For example, the left two canonical structures from Figure 2.2 are represented in the system as shown below

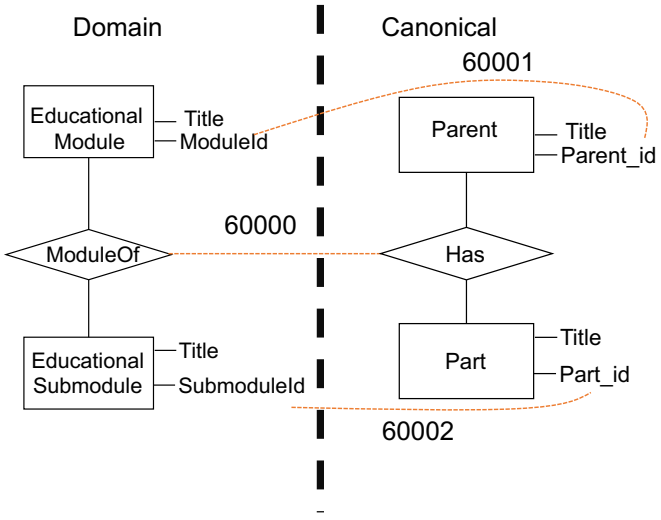


Figure 2.5: A mapping is shown between the parent-part canonical structure and the educational module domain structure.

in Result Set 2.4.

Result Set 2.4

```
# select * from cs;
      id      |              crs
--------------+-----------------------------------------
    ItemCS    | {(Item,ItemId,{ItemId,Title,Attribute})}
 ParentPartCS | {(Parent,ParentId,{ParentId,Title}),
              |  (Part,PartId,{PartId,Title}),
              |  (Has,HasId,{HasId,ParentId,PartId})}
```

Mappings can then be created between domain structures and canonical structures, such as that shown in Figure 2.5, where each mapping is comprised of a set of correspondences. Here, there is a mapping between the "ModuleOf" domain relation and the "Parent-Part" canonical relation. The mapping consists of a set of correspondences between the relationships and the "ModuleId" domain attribute and "parent_id" canonical attribute and the "SubmoduleId" domain attribute and "part_id" canonical attribute.

Mappings between canonical and domain structures are defined similarly to mappings between domain structures and local databases.

Domain Structure - Canonical Structure Mappings:

$cs\_ds\_m(\underline{id}, dsid, csid, cr\_dr\_ms(id, dr, cr, corrs(id, da, ca)))$

The set of all domain-structure-to-canonical-structure mappings is defined in the *cs_ds_m* nested relation. A mapping is defined as one tuple in the relation with an identifier *id*, the domain structure in the mapping *dsid*, the canonical structure in the mapping *csid*, and a nested relation of the relations mapped *cr_dr_ms*. Each relation mapping tuple in *cr_dr_ms* consists of an identifier *id*, the domain relation *dr*, the canonical relation *cr*, and a nested relation of the correspondences between domain and canonical attributes. Each correspondence consists of an identifier *id*, the domain attribute name *da*, and the canonical attribute name *ca*. Unlike local-to-domain mappings, domain-to-canonical mappings do not allow predicates in the correspondences. The mapping from Figure 2.5 is shown below in Result Set 2.5.

Result Set 2.5

```
# select * from cs_ds_m;
 id  |     dsid     |    csid    |                      cr_dr_ms
-----+--------------+------------+-----------------------------------------------------
 600 |EducationalDS|ParentPartCS| (6000,ModuleOf,Has,{(60000,ModuleOfId,HasId),
                                                      (60001,ModuleId,ParentId),
                                                      (60002,SubmoduleId,PartId)})
```

2.2 ALLOWED MAPPINGS

One of our main goals is to be able to facilitate non-technical content authors to be able to create their own mappings from their local databases to domain structures. To that end, we limit the types of mappings that are allowed in our system. In this section we describe the types of mappings that can be created in our system and compare how they relate to mappings described with traditional tuple-generating dependencies (TGDs) [36, 57].

The definition of a tuple-generating dependency is as follows:

$\forall x(\phi_S(x) \rightarrow \exists y(\psi_T(x, y))$

where ϕ and ψ are conjunctions of atomic formula over the source S and target T schemas respectively. This definition allows for arbitrarily complex mappings between source and target schemas. We next present the different forms of limited mappings we allow and show how they are expressed as TGDs.

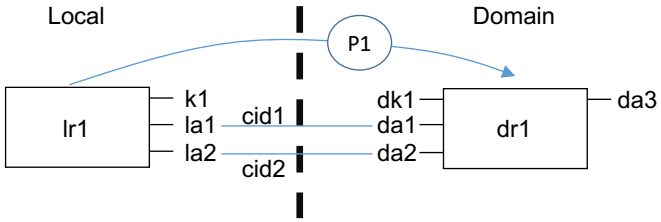


Figure 2.6: An example mapping between the "lr1" local relation and the "dr1" domain relation with predicate "P1", where there are correspondences between local attirbute "la1" and domain attribute "da1" (with id "cid1") and local attribute "la2" and domain attribute "da2" (with id "cid2") .

2.2.1 Local to Domain Mappings

In the general case, a local-database-to-domain-structure mapping in our system consists of a set of local-relation-to-domain-relation mappings, each of which is comprised of a set of correspondences. For example suppose we have the local database and domain structure shown in Figure 2.6 and below in Result Set 2.6.

Result Set 2.6

```
ldb(id,lrs(name,key,attrs))
---------------------------
ldb1 | {lr1,k1,{k1,la1,la2}}

ds(id,drs(name,key,attrs))
---------------------------
ds1 | {dr1,dk1,{dk1,da1,da2,da3}}
```

The local database, $ldb1$, has a single local relation, $lr1$, which has a key attribute, $k1$, and two non-key attributes, $la1$ and $la2$. The domain structure, $ds1$, has a single domain relation, $dr1$, which has a key attribute, $dk1$, and three non-key domain attributes, $da1$, $da2$ and $da3$. Consider the local-database-to-domain-structure mapping as shown in Figure 2.6 and below in Result Set 2.7.

Result Set 2.7

```
ds_ldb_m(id,ldbid,dsid,dr_lr_ms(id,lr,dr,p,corrs(id,la,da)))
-----------------------------------------------------------
dsldbm1 | ldb1 | ds1 | {mid1,lr1,dr1,P1,{cid1,la1,da1},
                                        {cid2,la2,da2}}
```

This local-database-to-domain-structure mapping exists between the local database, $ldb1$, and the domain structure, $ds1$, described above in Result Set 2.6,

and contains a single local-relation-to-domain-relation mapping with mapping id $mid1$. This local-relation-to-domain-relation mapping has the predicate $P1$ and contains two correspondences. The first, with id $cid1$, is between the $la1$ local attribute and the $da1$ domain attribute. The second, with id $cid2$, is between the $la2$ local attribute and the $da2$ domain attribute.

This mapping is then described in the TGD shown below.

$$\forall k1, la1, la2 \ \mathrm{lr1}(k1, la1, la2) \wedge P1 \rightarrow \mathrm{dr1}(GK(k1, ldb1, lr1, mid1), la1, la2, \text{`NULL'})$$

The function GK is used to generate a key for each tuple in the domain relation based on the key of the local relation, the local database, the local relation, and the mapping of which it is a part[2]. This key uniquely identifies the tuple from the local database and local relation from which the domain-relation tuple is derived. Key generation is done automatically creating an implicit mapping between the key of the local relation and the key of the domain relation. This TGD states that for every tuple in the local relation $lr1$, if the predicate $P1$ is true, there will be a corresponding tuple in the domain relation $dr1$ that has its domain-relation key defined by the function GK, has the local attribute $la1$ value for the domain attribute $da1$, has the local attribute $la2$ value for the domain attribute $da2$, and has the null value for the domain attribute $da3$ since $da3$ does not have a correspondence in this mapping.

In general the form of the TGDs that represent our allowed mappings in our system must conform to the following rules:

- The left-hand side of the implication can only contain a single local relation and a predicate (that only contains attributes from the local relation).
- The right-hand side can only have atoms from a single domain relation (there

[2]This key-generation function will be described in detail in the following chapter.

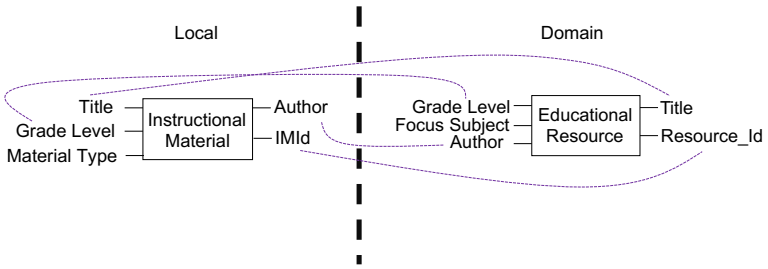


Figure 2.7: A straightforward mapping where each local and domain attribute only exists in a single correspondence. Correspondences ids are left out of this (and the following) figure since they are not needed to explain the TGD creation process (they would still be created automatically by the system).

may be more than one atom from the same domain relation which will be discussed below).

- There cannot be any existential variables on the right-hand side of the implication. All right-hand side variables must be from the left-hand side or the literal 'NULL' value.

We next describe how a TGD is constructed in our mapping system through the four basic cases allowed in our system and their combinations. A mapping is well-formed if it is one (or a combination of any) of the four base cases.

2.2.2 Straightforward Mappings

Straightforward mappings are mappings where each correspondence references unique domain and local attributes. For example, Figure 2.7 shows a straightforward mapping between the "Educational Resource" domain entity and the "Instructional Material" local entity. Each mapped domain attribute is in a single correspondence with one local attribute. Note that not all domain or local attributes need be included in the mapping; for example, neither the "Focus Subject" domain attribute nor the "Material Type" local attribute are included in this mapping.

Based on the definition of the local and domain relations:

InstructionalMaterial($IMId, Author, Title, GradeLevel, MaterialType$)

EducationalResource($ResourceId, Author, Title, GradeLevel, FocusSubject$)

we construct the TGD such that the local relation is on the left-hand side of the implication, all local variables are universally quantified, the key attribute of the domain relation is replaced by the key generating function GK[3], all other domain variables that exist in correspondences are replaced by their corresponding local variables, and all unmapped domain variables are replaced by the literal 'NULL'. A TGD that defines this mapping is as follows:

$\forall i, a, t, g, m$ InstructionalMaterial$(i, a, t, g, m) \rightarrow$

$$EducationalResource(GK(i), a, t, g, \text{'NULL'})$$

In this case, where no predicate has been defined for the mapping, the left-hand side of the implication contains only the local relation.

2.2.3 One-Local-Attribute-to-Many-Domain-Attributes Mappings

We also allow the case where the same local attribute may be in correspondences to multiple domain attributes[4]. Figure 2.8 shows this where there is a local entity, "Office" that represents an office branch for a company. It has a name, a manager, and a city where the branch is located. On the right side of the figure there is a domain entity, "Company", that represents an entire company and has attributes for the company name, a branch office, a manager, and its location. In this case, the city attribute of the local "Office" entity signifies both its branch and its location so there are two correspondences between the local "City" attribute and the "Branch" and "Location" domain attributes. The domain and local relations

[3]Note, we simplify the representation of GK here (and the rest of the section) by omitting the additional input parameters of the local database, local relation, and mapping id.

[4]We separate this case from the straightforward case above to explicitly show what is allowed in our system even though these two cases are technically quite similar.

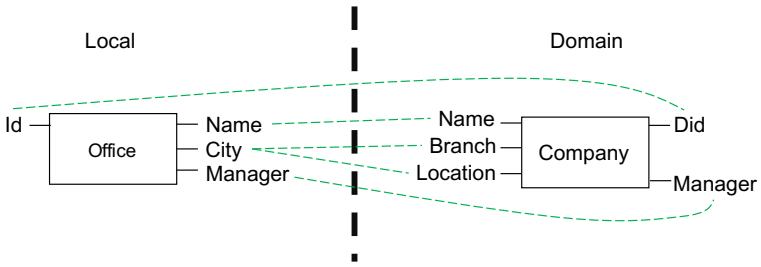


Figure 2.8: One local attribute to many domain attributes mapping where the "City" local attribute has correspondences to both the "Branch" and "Location" domain attributes.

are defined as follows:

$$\text{Office}(Id, Name, City, Manager)$$

$$\text{Company}(Did, Name, Branch, Location, Manager)$$

The mapping is then represented by the following TGD.

$$\forall i, n, c, m \ \text{Office}(i, n, c, m) \rightarrow \ \text{Company}(GK(i), n, c, c, m)$$

In this case, any local attribute that has correspondences to multiple domain attributes will be represented by the local variable being repeated in the domain relation for every correspondence. In this case, the "Branch" and "Location" domain attributes will get the value for the local "City" attribute.

2.2.4 Conditional Mappings

In the definitions above we mentioned that predicates may be attached to a mapping. These types of mappings are often needed when only a subset of the local data should appear in a domain relation. For example, Figure 2.9 shows the case where the "YouthAthlete" domain entity represents athletes eighteen years old or younger. In order to create a mapping between a local "Player" entity that may contain athletes of all ages, we need to specify that only athletes eighteen years old or younger should be in the mapping, we add predicates to the mapping by adding the directional line from the local relation to the domain relation with the

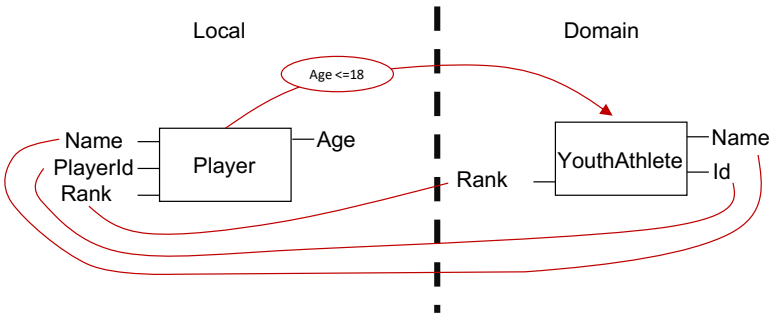


Figure 2.9: A conditional mapping where the predicate "Age<=18" has been added to the mapping.

desired predicate. The domain and local relations are defined as follows:

Player$(PlayerId, Name, Rank, Age)$

YouthAthlete$(Id, Name, Rank)$

The mapping is then represented by the following TGD.

$\forall p, n, r, a$ Player$(p, n, r, a) \wedge (a <= 18) \rightarrow$ YouthAthlete$(GK(p), n, r)$

In this case, all the variables are mapped like in the straightforward case but there is now the addition of the predicate to the left-hand side of the implication. Note that the predicate must be well formed and only reference attributes from the single local relation in the mapping.

2.2.5 Many-Local-Attributes-to-One-Domain-Attribute Mappings

Beyond the straightforward case, we also allow mappings where there exists multiple correspondences from different local attributes to a single domain attribute. For example, Figure 2.10 shows a mapping between the "DigitalLibraryResource" domain entity and the "AltInstructionalMaterial" local entity. The "DigitalLibraryResource" domain entity represents a case in which resources in a digital library have some fixed data attributes and then may have a number of metadata values to facilitate searching and indexing but that do not have their own attributes. In this case, the "AltIMId", "Author", and "Title" local attributes have

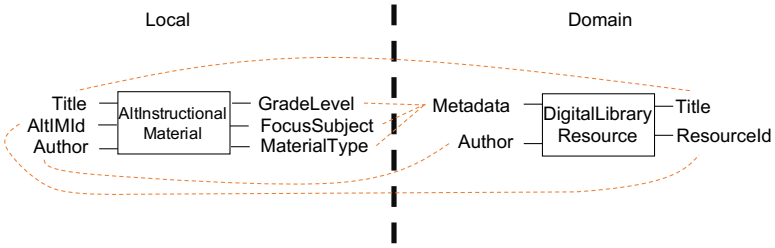


Figure 2.10: A many local to one domain mapping where the "GradeLevel", "FocusSubject", and "MaterialType" local attributes all have correspondences to the "Metadata" domain attribute.

straightforward correspondences to the domain entity. But, the "GradeLevel", "FocusSubject", and "MaterialType" attributes all have correspondences to the "Metadata" domain attribute. The local relation and domain relation are defined as follows:

AltInstructionalMaterial*(AltIMId,Author,Title,GradeLevel,Subject,MatType)*

DigitalLibraryResource*(ResourceId,Author,Title,Metadata)*

The mapping is then represented by the following TGD.

$$\forall aid, a, t, g, s, m \text{ AltInstructionalMaterial}(aid, a, t, g, s, m) \rightarrow$$
$$\text{DigitalLibraryResource}(GK(aid), a, t, g)$$
$$\wedge \text{ DigitalLibraryResource}(GK(aid), a, t, s)$$
$$\wedge \text{ DigitalLibraryResource}(GK(aid), a, t, m)$$

This TGD states that for every tuple in the local "AltInstructionalMaterial" relation there will be three tuples in the "DigitalLibraryResource" domain relation. One tuple for the domain relation will be added for each local attribute that is mapped to the same domain attribute[5]. For each instance of the domain relation in the TGD, the "ResourceId", "Author", and "Title" domain attributes are defined

[5]Note, this represents a typical DB *unpivot* operation, therefore, *GK(aid)* will not be a key for the resulting tuples in the relational model. Chapter 3 shows how this key will be maintained using our operators in the nested relational model.

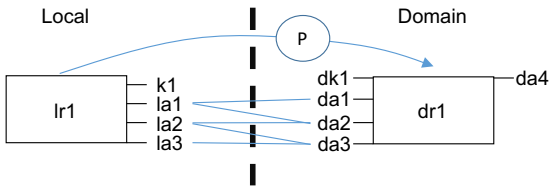


Figure 2.11: A local to domain mapping that combines all of the above cases.

as above in the straightforward case.

2.2.6 Combinations and Algorithmically Building TGDs

We now consider how these base cases can be combined into any allowable mapping in our system and show an algorithm for creating the appropriate TGD based on the mappings. Consider the mapping shown in Figure 2.11. This mapping combines all of the previous cases, it has a predicate P, a straightforwardly mapped domain attribute $da1$, two multiply mapped local attributes $la1$ and $la2$, and two multiply mapped domain attributes $da2$ and $da3$.

Algorithm 2.1 shows how we build TGDs generally. We will explain the algorithm using the example from Figure 2.11. The local relation and domain relation are shown below in Result Set 2.8 and the mapping is shown in Result Set 2.9.

Result Set 2.8

```
ldb
 id   |  lrs(name,key,attrs)
------+--------------------------
ldb1 | {lr1,k1,{k1,la1,la2,la3}}

ds
 id |      drs(name,key,attrs)
----+---------------------------------
ds1 | {dr1,dk1,{dk1,da1,da2,da3,da4}}
```

Result Set 2.9

```
ds_ldb_m
    id   |ldbid |dsid |  dr_lr_ms(id,lr,dr,p,corrs(id,la,da))
--------+------+-----+-------------------------------------
dsldbmC | ldbC | dsC | {mid1,lr1,dr1,P, {cid1,la1,da1},
                                          {cid2,la1,da2},
                                          {cid3,la2,da2},
                                          {cid4,la2,da3},
                                          {cid5,la3,da3}}
```

Following the algorithm we begin by creating the left hand side of the TGD in lines 3 to 9

Algorithm 2.1 Algorithm for building TGDs from local to domain mappings.

```
1: procedure BuildTGD(dr_lr_m, lr, dr)
2:     ▷ Where dr_lr_m, lr, and dr are the nested relations defined as dr_lr_m(mid,
   lr, dr, P, corrs(cid, la, da)), lr(name, key, attrs), dr(name, key, attrs)
3:     lhs ←'∀'
4:     lr ←' lr.name('
5:     for all la in lr.attrs do
6:         lhs ← lhs+'la,'
7:         lr ← lr+'la,'
8:     end for
9:     lhs ← lhs + lr+')'
10:    if dr_lr_m.P exists then
11:        lhs ← lhs+'∧'+dr_lr_m.P
12:    end if
13:    da_corrs ← dictionary()
14:    for all da in dr.attrs do
15:        da_corrs[da] = set()
16:    end for
17:    for all corr in dr_lr_m.corrs do
18:        da_corrs[corr.da].append(corr.la)
19:    end for
20:    for all da in dr.attrs do
21:        if length(da_corrs[da]) == 0 then
22:            da_corrs[da].append('NULL')
23:        end if
24:    end for
25:    da_corrs[dr.key].append('GK('+lr.key+')'))
26:    dr_instances ← list(list())
27:    for all da in dr.attrs do
28:        temp_list ← list(list())
29:        for all instance in dr_instances do
30:            for all la in da_corrs[da] do
31:                temp_instance ← instance
32:                temp_list.append(temp_instance.append(la))
33:            end for
34:        end for
35:        dr_instances = temp_list
36:    end for
```

Algorithm 2.1 *Continued from previous page*

```
37:     for all instance in dr_instances do
38:         rhs ←'dr.name(instance)∧'
39:     end for
40:     return lhs+'→'+rhs
41: end procedure
```

$\forall k1, la1, la2, la3$ lr1$(k1, la1, la2, la3)$.

Then in lines 10 to 12 we add the predicate if there is one, in this case "P", so now we have

$\forall k1, la1, la2, la3$ lr1$(k1, la1, la2, la3) \wedge P$.

We create a dictionary of sets to represent all the correspondences indexed by the domain attributes in lines 13 to 19. In this case we create the dictionary below.

```
da_corrs[dk1]={}
da_corrs[da1]={la1}
da_corrs[da2]={la1,la2}
da_corrs[da3]={la2,la3}
da_corrs[da4]={}
```

For any non-key domain attribute that is not in any correspondence we then add 'NULL' (lines 20 to 24). The entry for $da4$ then becomes

```
da_corrs[da4]={'NULL'}.
```

We add the implicit key generating mapping in line 25.

```
da_corrs[dk1]={GK(k1)}
```

We then do a recursive list comprehension of the correspondences to create the lists of variables for the right hand side atoms of the TGD (lines 26 to 35) which produces the following list of lists.

```
dr_instances=((GK(k1),la1,la1,la2,'NULL'),
              (GK(k1),la1,la2,la2,'NULL'),
              (GK(k1),la1,la1,la3,'NULL'),
              (GK(k1),la1,la2,la3,'NULL'))
```

Then we create the right hand side in lines 36 to 38 which creates

```
dr1(GK(k1),la1,la1,la2,'NULL')∧
dr1(GK(k1),la1,la2,la2,'NULL')∧
dr1(GK(k1),la1,la1,la3,'NULL')∧
dr1(GK(k1),la1,la2,la3,'NULL')
```

Lastly, we combine the left hand and right hand sides in line 39 resulting in the following TGD.

$$\forall k1, la1, la2, la3 \text{ lr1}(k1, la1, la2, la3) \wedge P \rightarrow dr1(GK(k1), la1, la1, la2, \text{`}NULL\text{'}) \wedge dr1(GK(k1), la1, la2, la2, \text{`}NULL\text{'}) \wedge dr1(GK(k1), la1, la1, la3, \text{`}NULL\text{'}) \wedge dr1(GK(k1), la1, la2, la3, \text{`}NULL\text{'})$$

2.2.7 Domain-to-Canonical Mappings

We assume, generally, that domain structures will map fairly directly onto canonical structures and therefore require that a mapping will always be of the straightforward case described above. We expect that a domain structure and a canonical structure will often be isomorphic, differing only in their names. The algorithm to produce TGDs is shown in Algorithm 2.2.

Up through line 22, this algorithm is essentially the same as the local-to-domain algorithm minus the predicate. In line 22, the key from the domain relation is automatically mapped to the canonical relation but retains the value generated by GK. Then the right hand side is created in lines 23 to 27. Since the only case that is possible is the straightforward case, there can be at most one correspondence for each attribute and only a single atom on the right hand side, so we do not need to use the list comprehension from the previous algorithm.

Algorithm 2.2 Algorithm for building TGDs from domain to canonical mappings.

```
1: procedure BuildTGD(cr_dr_m, dr, cr)
2:     ▷ Where cr_dr_m, dr, and cr are the nested relations defined as
   cr_dr_m(id,dr,cr,corrs(id,da,ca)),dr(name,key,attrs),cr(name,key,attrs)
3:     lhs ←'∀'
4:     dr ←' dr.name('
5:     for all da in dr.attrs do
6:         lhs ← lhs+'da,'
7:         dr ← lr+'da,'
8:     end for
9:     lhs ← lhs + dr+')'
10:    ca_corrs ← dictionary()
11:    for all ca in cr.attrs do
12:        ca_corrs[ca] = set()
13:    end for
14:    for all corr in cr_dr_m.corrs do
15:        ca_corrs[corr.ca].append(corr.da)
16:    end for
17:    for all ca in cr.attrs do
18:        if length(ca_corrs[ca]) == 0 then
19:            ca_corrs[ca].append('NULL')
20:        end if
21:    end for
22:    ca_corrs[cr.key].append(dr.key)
23:    rhs ←'cr.name('
24:    for all ca in cr.attrs do
25:        rhs ← rhs + ca_corrs[ca]+','
26:    end for
27:    rhs ← rhs+')'
28:    return lhs+'→'+rhs
29: end procedure
```

2.3 USER STUDY

Here we present the results of our user study where seven departmental staff at our university (responsible for creating content on the university web site) with a range of technical expertise, were asked to provide mappings. The study was performed with human subjects approval from our university institutional review board. Subjects were provided a short training session and then required to use our system to create mappings for a widget in a website that they had not seen before. The overriding goal of the study was to evaluate the feasibility of our mapping approach through discount usability testing [59] where the usability (effectiveness, efficiency, and satisfaction) of a system is tested through small and simple scenarios that lead to quick feedback but are not statistically significant. Our goal is to show that our system is usable but since we are not user interface developers, we do not expect nor attempt to evaluate the user interface of our mapping system.

2.3.1 Design of the User Study

We designed our user study to test whether subjects could create mappings between local schemas and domain structures. Subjects were asked to complete three tasks; one training task in which participants were guided through the process of creating mappings in a site with a fairly simple schema; and, two tasks where participants worked on their own to create mappings in two different sites (one with a simple schema and one with a more complex schema).

Subjects were given a demographic questionnaire at the beginning of the session, evaluation questionnaires at the end of each of the two testing tasks, and an overall evaluation questionnaire at the end of the session. As we expect our tool to be used by domain-savvy users, our test was limited to sites in a single domain (in this case, an educational domain) and our users were chosen from staff of different departments within our university that are responsible for updating

their departmental web pages.

For the training task, subjects used a website built using the library local schema shown in Figure 2.13. This schema has a simple (one-directional) hierarchy between "Library", "Collection", "Book", "Chapter", and "Section". There is also a bidirectional relationship between "Category" and "Book" so that the subjects could create recursive mappings using the tool. The subjects were shown the structure of the site using only the hypertext links in the webpages within the operational website (shown in Figure 2.12).

The goal of the training task was to show the subjects how to create different instances of the navigation widget shown in Figure 2.14. The widget is a hierarchical navigation widget that allows a structured website to be shown in a hierarchical tree. The figure shows an instance of a library with two collections, where each collection has a number of books, and each book has a number of chapters. In this case the "Category"-"Book" relationship is not mapped, therefore no categories appear in this instance of the widget.

The subjects were then shown the mapping tool (Figure 2.15) that, for a given domain structure, allows the user to select a content type from a list of all possible content types in the site and then choose a relationship associated with that type. Part of the mapping interface is a preview widget that shows how the navigation widget in the site would look using the given mappings. The tool then allows users to delete a specific mapping, save all of their mappings, or delete all of their mappings (Figure 2.16).

In the training tasks, the subjects were asked to create a number of specific mappings and encouraged to make additional mappings, as desired. There may be many different mappings that can be created within any given site for any number of reasons, so we explicitly allowed our subjects to create whatever mappings they felt were appropriate. Since the choice of mappings is subjective, we did not test to see if subjects would create any specific mappings. Mappings were

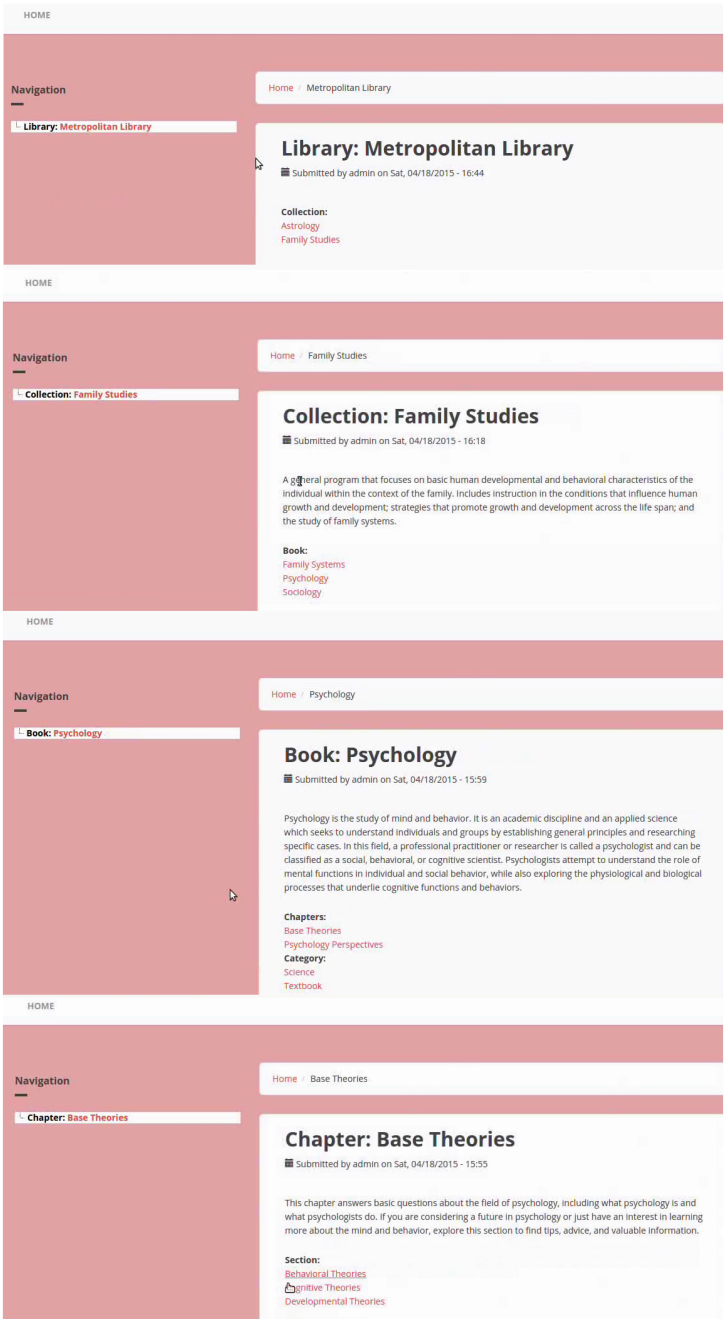


Figure 2.12: The training task website showing a library, a collection, a book, and a chapter webpage. Links to related content types are in red inside the white boxes.

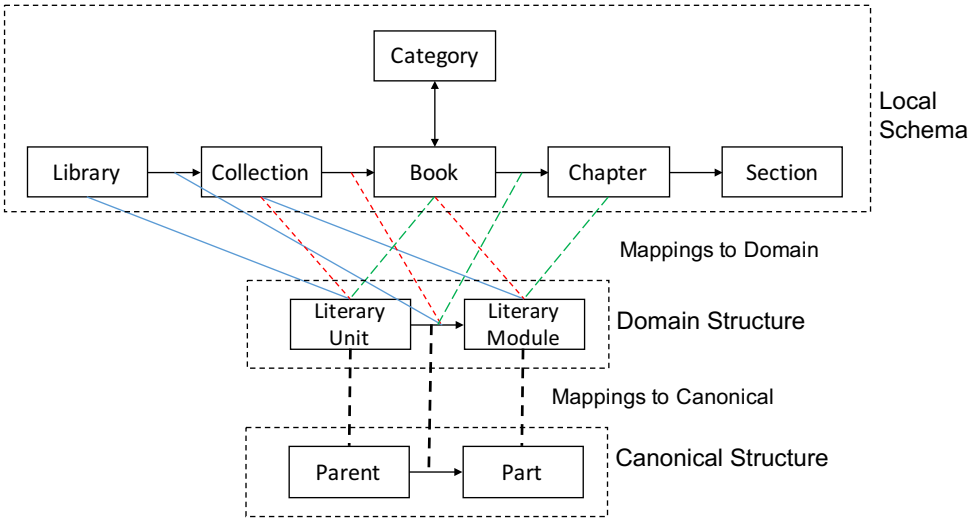


Figure 2.13: A local library schema (top, shown in a simplified ER diagram that only has enitities and has directional arrows representing the links in the website), the library domain structure (middle), the parent-part canonical structure (bottom), and mappings between the three.

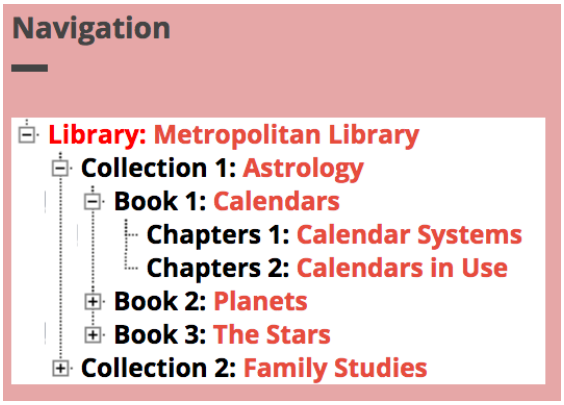


Figure 2.14: An instance of the navigation widget used in the training task of the user study with mappings between the "Library-to-Collection", "Collection-to-Book", and "Book-to-Chapter" local relationships and the "Literary-Unit-to-Literary-Module" domain relationship.

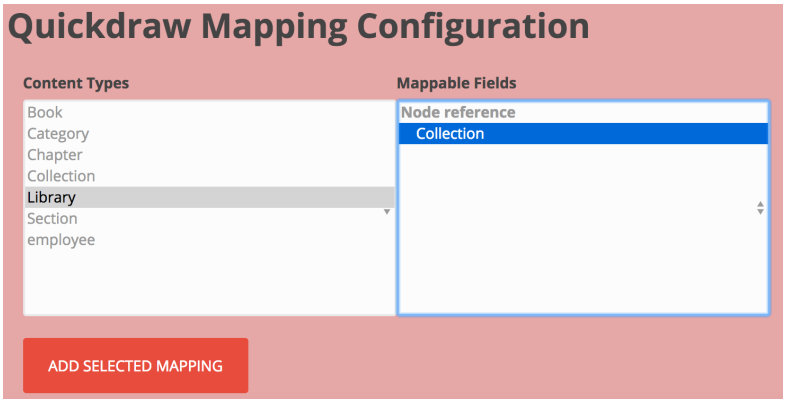


Figure 2.15: In the mapping interface, a user first selects the domain relation to which they want to create a mapping from a dropdown list (not shown here). Then, a user selects a content type (on the left) and then is shown all possible relationships to other content types (on the right). Here, a mapping is created between "Library-to-Collection" local relationship and the "Literary-Unit-to-Literary-Module" domain relationship.

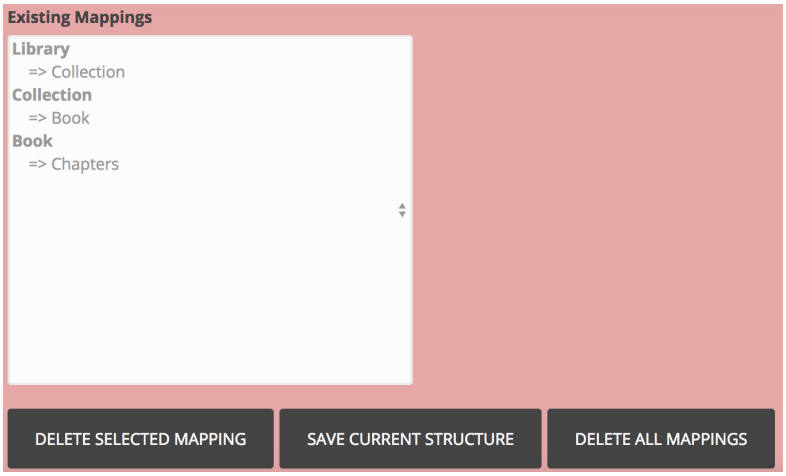


Figure 2.16: Three mappings are shown: between the "Library-to-Collection", "Collection-to-Book", and "Book-to-Chapters" and the "Literary-Unit-to-Literary-Module" domain relationship. Users can select a specific mapping to delete, save the entire set of mappings, or delete the entire set of the mappings.

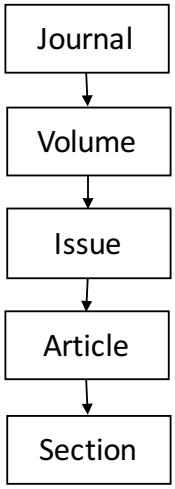


Figure 2.17: Schema for first task in the study (shown in a simplified ER diagram that only has enitities and has directional arrows representing the links in the website).

only deemed incorrect if the end result produced irregular widget behavior (i.e., duplicate mappings or disjoint mappings). After the scripted part of the training session, participants were allowed to explore the training site and the mapping tool for as long as they desired.

After the training task, participants were then asked to create mappings they saw as appropriate for a website with the schema shown in Figure 2.17. This schema is a simple hierarchy of an academic journal using unidirectional relationships only. For the second testing task, participants were asked to create mappings they saw as appropriate for a website based on a university schema shown in Figure 2.18. The schema includes bidirectional relationships and cycles.

Participants for the study were recruited from the pool of departmental administration staff from the university who are in charge of updating the university webpage for their respective departments. All participants had working knowledge of Journals, Libraries, and Universities. The participants had varying degrees of technical expertise ranging from three to more than ten years of website configuration experience and none to more than ten years of database experience.

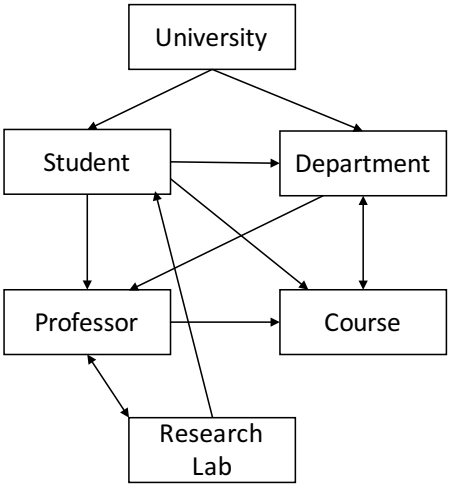


Figure 2.18: Schema for second task in the study.

2.3.2 User Behaviors

We showed participants how to browse the site (based on the schema in Figure 2.13), see a preview of the widget, and create large and small mappings. We emphasized the use of the preview functionality as we believed it would aid the creation and checking of mappings.

Figure 2.20 shows an overview of the study subjects' sessions. Figure 2.19 shows the legend for the color codes shown in Figure 2.20 and the following figures in this section. Each subject's session is represented in a horizontal bar beginning with their anonymous id. The sessions are broken into boxes for each task, the first (pink) box shows the training task, the second (blue) box shows the first testing task, and the last (green) box shows the second testing task. The smaller boxes inside each box represent the various actions performed by the subject within the task (explained below). The longest session lasted a little less than 50 minutes and the shortest was less than 20 minutes. This variation is unsurprising given the open-ended nature of the tasks. In most sessions, subjects took a longer time with the second task, likely due to the more complex nature of the local schema for the site in that task. For the two subjects who completed the second task in

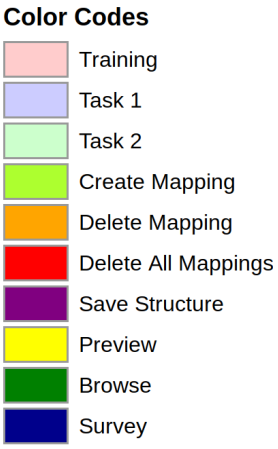


Figure 2.19: Legend for the color coding shown in Figures 2.20, 2.21, 2.22, 2.23, and 2.24.

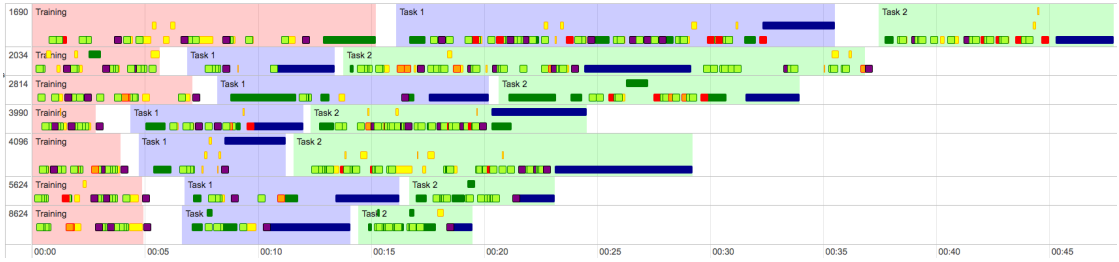


Figure 2.20: Timelines of user sessions showing length of training, Task 1, and Task 2. The x-axis shows session time, the longest session lasting 50 minutes.

the shortest times, one created a single set of mappings for the university, without cycles, and decided they were done while the second appeared to rush through the task and saved a set of mappings that included duplicate mappings. This was the only subject to save a set of mappings we deemed to be incorrect; other subjects created structures that had duplicate mappings or contained disjoint parts of the schema but in all cases these subjects discovered and deleted their bad mappings.

In Figure 2.21 all of the timelines have been scaled up to an hour in length so that we can compare the relative amount of time each user spent on each task. As each task (including the training task) was open ended, the amount of time spent

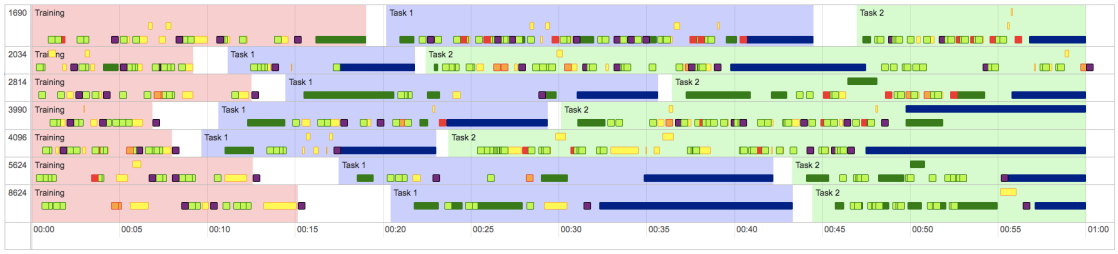


Figure 2.21: Normalized timelines (where all sessions are stretched to an hour in length) of user sessions showing the comparative length of training, Task 1, and Task 2.

in each task varied greatly.

As mentioned above, we believed that the preview feature of our tool would be useful for checking and evaluating mappings. We found that some of our participants chose to use the preview feature while others chose to browse through the live site and see the live widget in the context of the actual webpages. Figure 2.22 shows an example of the use of the preview. In this case the subject starts the task by creating six mappings, uses the preview function to check the mappings, deletes two mappings, uses the preview again, and then saves the set of mappings. This subject continues creating a few mappings and checking with preview. Figure 2.23 shows another example of a previewer where the user starts by creating a single mapping, previews that mapping, and then saves the set of mappings. This user continues previewing after each new mapping. Contrast the preview behavior with Figure 2.24, where the subject creates five mappings, browses the site, creates two more mappings, browses the site again, creates six more mappings, and then saves the set of mappings.

Figures 2.22 and 2.23 also demonstrate two other behavior patterns observed in the test. In Figure 2.22, the subject creates a few mappings, checks them with preview, then deletes a mapping or two before saving the set of mappings. Compare that with Figure 2.23, where the subject often creates a number of mappings and when they decide that they are incorrect or not to their liking they delete the

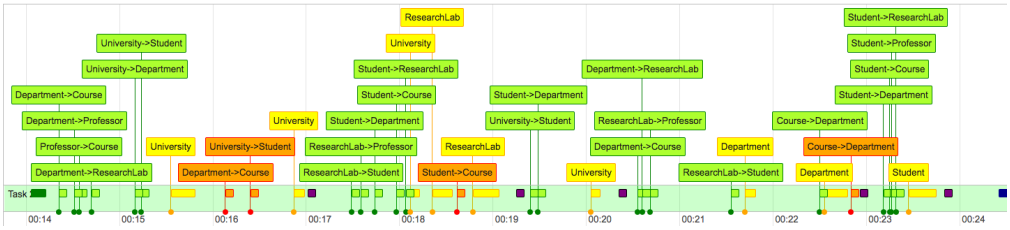


Figure 2.22: A study session of Task 2 demonstrating the preview, test and check, and the entity-centric behaviors.

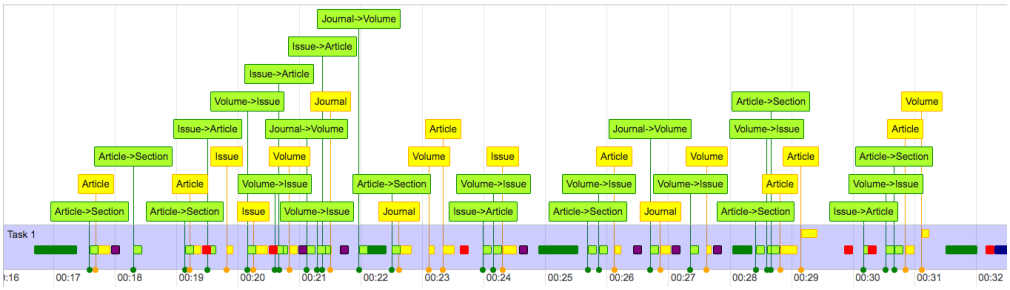


Figure 2.23: A study session of Task 1 demonstrating the preview, delete and start over, and the random behaviors.

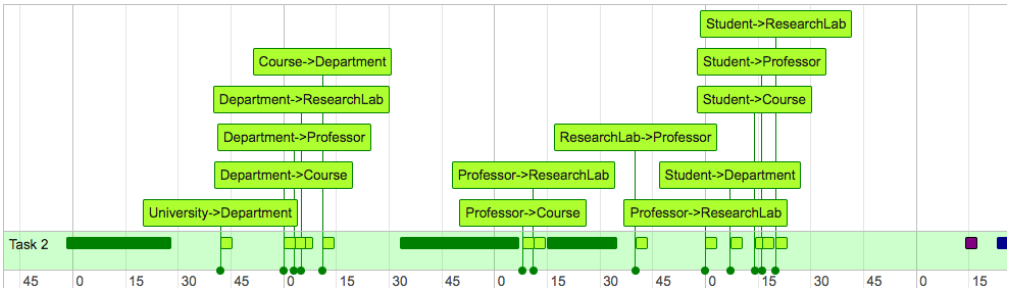


Figure 2.24: A study session of Task 2 demonstrating the browse, large, and entity-centric behaviors. Darker rectangles represent browsing behavior, mappings are displayed above the lighter colored squares, and the dark square at the end saves all mappings.

entire set of mappings and start over.

The three examples shown thus far also demonstrate the two different ways subjects approached the process of mapping. In Figures 2.22 and 2.24 the subjects created larger navigational structures (with more mappings). These structures were also built in an entity-centric way, where the subject starts at one type, created all the mappings related to that type and then moved to the next. Contrast that to Figure 2.23, where the subject creates many small structures and the mappings are often created in what appears to be a random fashion.

2.3.3 Results

With regard to our main goal, the study showed that domain-savvy users can perform the mapping tasks using our system. All participants were able to complete the given tasks within a reasonable time frame (we designed the test to take no more than one hour with the caveat that the open-ended nature could have made it take longer) and, although they could leave at any time during the test, no one left the test prematurely. (One participant inadvertently failed to complete the final questionnaire, but completed all tasks and task questionnaires.) Participants were asked to rate the overall usefulness of the tool and enjoyment of mapping on a scale of one (Strongly Disagree) to five (Strongly Agree) with the average response for both questions being 3.7.

An interesting aspect of the behaviors listed above is how groups of differing behaviors corresponded to satisfaction results of the tool and mapping. Table 2.1 shows some of this detail. Though not statistically significant due to our sample size, we observed that the group of subjects that used preview was also the group with more than five years of experience. Those with less experience tended to exhibit more browsing behavior.

Overall those with less technical experience tended to find the system more useful than those with more experience. In the questionnaires related to the tasks,

Table 2.1: Aggregated user feedback showing satisfaction with the system for each task and overall and a scale of one (Strongly Disagree) to five (Strongly Agree).

	Satisfaction		
Behavior Group	Task1	Task2	Overall
Previewer and Experienced (n=4)	2.8	3.4	3.6
Non Previewer and Inexperienced (n=3)	5	2	4
Larger and Entity-centric (n=4)	4	3.25	4.7
Smaller and Random (n=3)	2.7	2.7	2.7

those participants with more experience expressed some frustration that they were limited to what the tool could do when they knew how to edit HTML directly to get the results they wanted. Also of note is that even though inexperienced users preferred the tool overall, they were less satisfied with the interface during the second more complex task.

Table 2.1 also shows the difference between the large mapping/entity-centric mappers and the small mapping and random mappers. We see that across the board, the larger and entity-centric mappers were more satisfied with the tool on each of the tasks and overall. It is likely that these subjects had a better understanding of the structure of the sites, our tool, and the tasks.

2.4 RELATED WORK

Our work on domain structures is based on the Entity-Relationship conceptual model [25, 79] (with a straightforward representation as a relational schema). Domain structures can be viewed as design patterns similar to those proposed in data modeling [8], the co-design and metastructure approach [51, 80], and ontology creation [37, 67]. Blaha [8] defines domain-independent (like canonical structures) *patterns* that compose generic data model constructs and domain-dependent *seed models* that can be used as a starting point for a schema (like domain structures). Similarly, ontology design patterns [67] represent domain-independent (like canonical) models that may be elaborated for a domain. In both of these approaches, the patterns are used to build new systems; existing systems are not mapped to these patterns. We approach pattern use in the opposite manner: we overlay patterns on existing systems in order to extend their functionality (by allowing the content author to place widgets in their site). Our domain structures can also be seen as abstract superclasses of the various local schema types to which the domain structures have been mapped, similar to view integration and view cooperation [79].

Generic schemas and functionality have been explored extensively in programming and data management, and bring with them many benefits. Generic schemas aid in development by allowing functions, code, and constraints to be defined generically and aid in the definition and creation of new (more complex) schemas and systems and allow for a greater reuse of schema [62]. Using generic schemas can provide faster development, even with complex models, while minimizing development complexity [62]. Generic types in programming languages such as Java [43] or C# [27] can provide common functionality to many different heterogeneous types. We take this approach (where our canonical schemas are generic schemas) and add the ease of use of schema mapping systems such as CLIO [57], to enable non-technical users to make use of generic functionality by inserting a user-friendly

schema (domain structures) between the generic (canonical) schema and the local schema.

Web development frameworks [33] also often provide a generic relational mapping to convert complex user-defined schemas into generic formats in their database back-ends. Often an instance of a content type created by a user in the web front-end is stored in the database with a table for each field of the object plus an instantiation of some base class. In contrast, Object-Relational Mappers (ORMs) [46] provide an algorithmic mapping between an object and a relational table that contains attributes for each of the fields in the object. Web-development frameworks can provide some basic generic functionality for building pages and websites, but more complex widgets are limited to predefined models (e.g., a typical calendar widget uses a single defined event type or an address-book widget has a predefined contact type). Our system brings generic widgets to the front-end of these systems, which accommodates the semantic heterogeneity of the underlying systems.

Conceptual models have been used as the basis for building Web information systems in projects such as WebML [24] and Araneus [4, 5, 54]. These approaches, like content management systems and web development frameworks, provide richly structured websites and facilitate the creation of these sites via modeling instead of coding. These approaches have many of the same constraints to generic functionality that web-development frameworks have. We could extend conceptual model-based information systems in the same fashion as we extend web-development frameworks.

We take inspiration from systems such as CLIO [57] for our mapping system. Like Clio, we want users to create mappings by simply drawing lines from local schemas to domain structures. In contrast, we expect users to map local schemas, perhaps many times, to our small domain structures instead of trying to create entire schema mappings to a single global schema. Domain-structure-specific mappings adds flexibility in how the domain structures may be later composed

and means that end users need not understand every domain structure that could be mapped (only the domain structures of interest to the user for a specific instantiation of a widget). Since we do not seek to fully integrate our local schemas, we avoid the problem of merging heterogeneous schemas present in data integration [48], model management [7], or ontology alignment [35].

The flexibility of our mappings is also inspired by pay-as-you-go data integration, such as that proposed by Madhavan [29]. Our widgets work with as many or as few mappings as are provided, allowing us to create widget previews that can then help users create further mappings.

Our work has been strongly influenced by work in schema mapping [32] and ETL [57] where users can draw simple lines between source and target schemas. While these tools are automated to facilitate the schema-mapping process, the mappings themselves and the tools to use them are targeted at expert database developers. We adapt this approach to allow non-expert users to do similar mappings, but limited to a very simple form.

We are also inspired by the field of end-user programming [44, 50] whose goal is to get non-developer users of software to create, modify, and extend that software. Current end-user web-programming paradigms [72, 85] focus on allowing end-users to create "mashups" of existing widgets and data on the Internet. While these "mashups" are useful for users of existing systems, modern web development tools have facilitated end-user website creation, which cannot leverage these tools without conforming to their pre-existing schemas. Our system brings end-user programming to the level of widget population through mapping, which can then be used in conjunction with existing web end-user programming.

Our domain and canonical structures can be viewed as a form of superimposed information [52] on top of local databases. Earlier work on superimposed information provide methods for uniform access to a variety of base information sources

(e.g., word processing documents, web pages, data files). Superimposed applications can then access the heterogeneous base information sources, for example building web mashups [58] from base sources such as web pages and PDFs (analogous to a widget built against a canonical structure in our system). We profit from having homogeneous base and superimposed models (the relational model) and avoid many of the challenges faced in a typical superimposed information scenario. Through the use of marks a superimposed information system can also determine the base context (base specific information such as font type and size in a word processing document or the row and column numbers of data from a spreadsheet) of information presented at the superimposed layer. In a similar fashion, we want to bring the context of our local databases through to the domain and canonical levels (where context in our system means local relation and attribute names).

2.5 CHAPTER SUMMARY

In this chapter we formally defined the various levels of schema within our system: local, domain, and canonical. We discussed how we relate to the current state of the art.

We defined how mappings can be created between the various levels. We demonstrated how these mappings work. We defined the types of mappings that we allow in our system and demonstrated how they compare to full tuple-generating dependencies.

We also provided the results from a user study of our mapping system using both technical and non-technical users. We show that all subjects of the study were able to successfully use the mapping system. We also show that users generally enjoyed using our system.

Chapter 3

QUERY LANGUAGE

In Chapter 2 we showed how domain and canonical structures can be defined to support domain-specific patterns and domain-independent generic patterns, respectively. In order to use these structures, we define a query language in this chapter that is able to return meaningful results by using queries that we generate against the local databases based on queries written for widgets that were originally expressed in the form of the canonical or domain structure. The query result retains the local database semantics (the attribute names). We do this by defining four new operators as an extension to the nested relational and relational algebras.

In this chapter we build on the formalism presented in Chapter 2 to define our new operators. We make the following contributions:

- We define the *apparent* and *underlying* models used within our systems for query and storage. These models allow developers to use the system at the canonical, domain, and local database levels as relational databases (the apparent model) while the internal system uses a nested relational format (the underlying model). We discuss the reason for each of these models and the definitions of the models in the next section.

- We define the *apply* (α) operator at the domain level, which creates corresponding queries against local databases that return integrated data from all mapped local databases in the nested relational form of the domain structure (i.e., in the underlying model). The *apply* operator is designed to work in a

number of different scenarios; we provide examples of typical cases for which it may be used.

- We define the *canonical apply* (θ) operator that is introduced into queries at the canonical level, which creates corresponding queries against domain structures that return integrated data from all mapped domain structures in the nested relational form of the canonical structure (i.e., in the underlying model). We provide an example to demonstrate the operator.

- We define the *apparent model* (κ) operator that provides a relational projection of the underlying model of a canonical or domain structure into the apparent model. We also provide an example of its use.

- We define the *type* (τ) operator that provides local type information to the canonical or domain level. This operator supports what we call local radiance because it allows the schema names from the local database to appear in widget output, for example.

- We define relational equivalences that can be used with our operators and show how they can be used to optimize performance in the implementation of our query interface.

- We evaluate the performance of our implemented query interface against hand-written integration queries as well as queries produced by widgets coded in a web development framework (Drupal [33]).

3.1 APPARENT AND UNDERLYING MODELS

From a query-writing perspective our goal is to maintain a familiar interface (i.e., the relational model) that allows developers to not worry about the complexities of our query interface, while benefiting from its power. We start with a small

example. Figure 3.1 shows two local databases, one describing football teams and another describing university tennis teams. The bottom of the figure shows the ER representation of the local schemas. The top of the figure shows the representation of the local relations in those schemas using the formalism from the previous chapter. Figure 3.2 shows subsets of the tennis and football databases. Each subset has a single entity, "Employee" and "Student" respectively, and they are each mapped to the "Person" entity in the domain structure (right side of figure). Local data is shown on the left side of the figure.

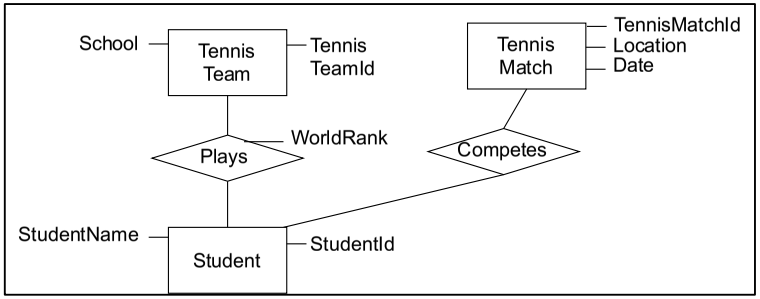
Our goal is to allow developers to use a query such as "`#select * from Person;`" and receive the result shown in Figure 3.3, which allows the developer to treat our system like any other relational database. We call this relational model interface in our system the *apparent model*.

From a system perspective, we need to retain more information than is shown in the apparent model, including local type information as well as the provenance of the mappings that populated the data. To do that our query interface uses the nested relational model as our *underlying model*. The output of the *apply* and *canonical apply* operators defined below are expressed in the nested relational model. For each tuple returned from a local database we first create a new identifier based on the local mappings and the local identifiers, and then for each domain or canonical attribute in the domain or canonical structure we define a nested attribute that contains the mapping and local type information, as shown in Figure 3.4. Specifically, an identifier will be generated with provenance information and then for each attribute a_i in the domain or canonical relation there will be a nested attribute of the form a_i(value, meta(mid, cid, type)), where *mid* is the mapping id, *cid* is the correspondence id, and *type* is the local attribute name.

The rest of this chapter will show how we use the underlying model to enable our query operators. We do not expect nor require end users or widget developers to use or understand the underlying nested relational model.

```
=>select * from ldb;
     id      |              lrs(name,lrid,attrs(name))
-------------+-----------------------------------------------------------------------
 TennisDB    | (Competes,CompetesId,{CompetesId,TennisMatchId,StudentId})
 TennisDB    | (TennisMatch,TennisMatchId,{TennisMatchId,Location,Date})
 TennisDB    | (TennisTeam,TennisTeamId,{TennisTeamId,School})
 TennisDB    | (Student,StudentId,{StudentId,StudentName})
 TennisDB    | (Plays,PlaysId,{PlaysId,StudentId,TennisTeamId,WorldRank})
 FootballDB  | (PlaysFor,PlaysForId,{PlaysForId,FootballTeamId,Position,Goals,EmployeeId})
 FootballDB  | (FootballTeam,FootballTeamId,{FootballTeamId,FootballTeamName})
 FootballDB  | (Employee,EmployeeId,{EmployeeId,EmployeeName,Address})
 FootballDB  | (Against,AgainstId,{AgainstId,FootballTeamId,FootballGameId})
 FootballDB  | (Manages,ManagesId,{ManagesId,FootballTeamId,EmployeeId})
 FootballDB  | (FootballGame,FootballGameId,{FootballGameId,City,Day})
```

TennisDB



FootballDB

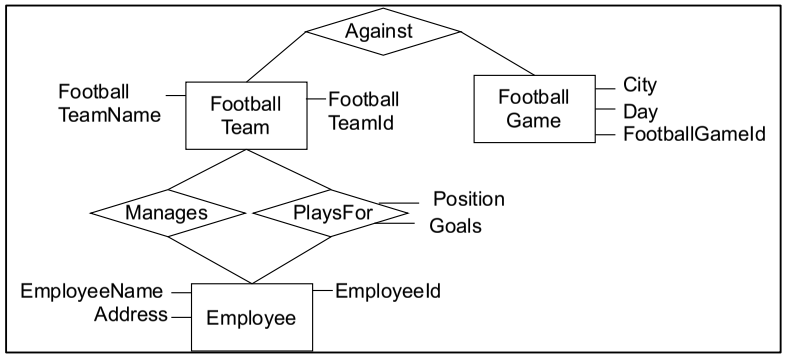


Figure 3.1: Local database schemas for the university tennis team (left) and football team (right).

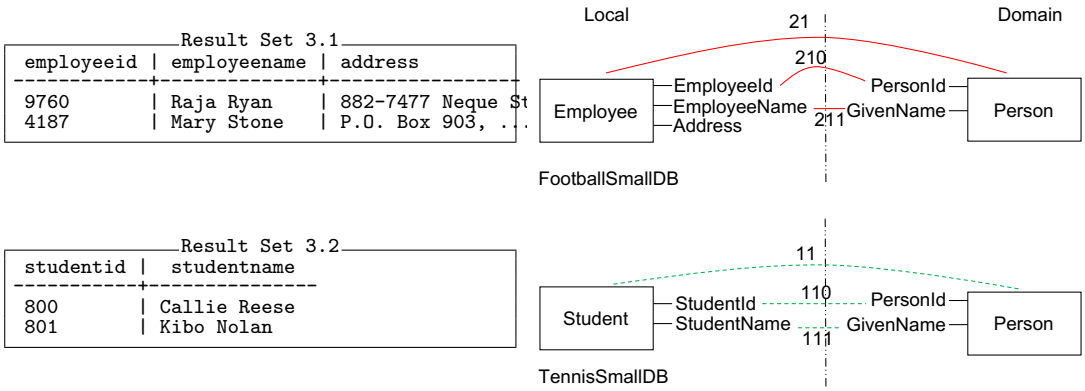


Figure 3.2: Data from the Employee and Student subsets of the football and tennis local databases (left). The Employee and Student local entities mapped to the Person entity in the domain structure (right).

Result Set 3.3

```
#select * from Person_Apparent;
personid   |  givenname
-----------+---------------
 9760      | Raja Ryan
 4187      | Mary Stone
 800       | Callie Reese
 801       | Kibo Nolan
```

Figure 3.3: Domain structure query result in the apparent model.

3.2 OVERVIEW

Figure 3.5 shows an overview of our query interface with its query language. The upper part of the figure shows examples of local-domain mappings and domain-canonical mappings, local schemas are on the left and domain structures are in the middle. Local to domain (DS-LDB) mappings are created between local and

Result Set 3.4

```
#select * from Person_Underlying;
               id          |personid(value,meta(mid,cid,type)| givenname(value,meta(mid,cid,type)
---------------------------+---------------------------------+-----------------------------------
FootballDB.Employee.21.9760|  {(9760,(21,210,EmployeeId))} |{(Raja Ryan,(21,211,EmployeeName))}
FootballDB.Employee.21.4187|  {(4187,(21,210,EmployeeId))} |{(Mary Stone,(21,211,EmployeeName))}
TennisDB.Student.11.800    |  {(800,(11,110,StudentId))}   |{(Callie Reese,(11,111,StudentName))}
TennisDB.Student.11.801    |  {(801,(11,110,StudentId))}   |{(Kibo Nolan,(11,111,StudentName))}
```

Figure 3.4: Domain structure query result in the underlying model.

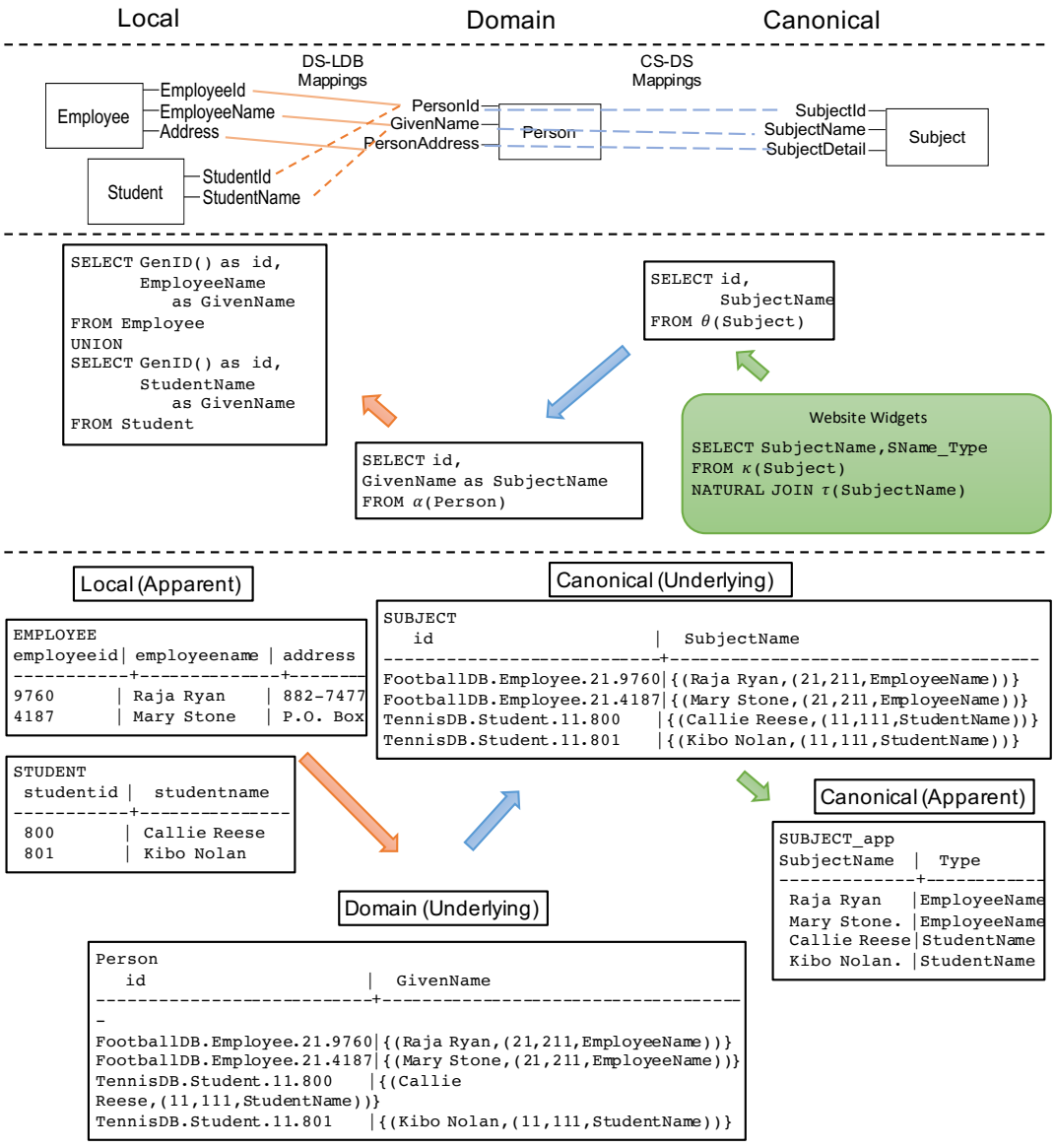


Figure 3.5: An overview of our query language and query interface for widgets. The top section shows the conceptual model of our query interface. The middle section shows our query operators at the various levels. The bottom section shows data at the four levels from the middle section. The local (far left) and canonical apparent (far right) levels are in the relational model while the domain underlying (middle left) and canonical underlying (middle right) levels are in the nested relational model.

domain structures. Canonical structures are on the right and domain-structure-to-canonical-structure (CS-DS) mappings are created between domain and canonical structures.

The middle section of Figure 3.5 shows how queries work in our query interface from the apparent canonical level back to the local database. Starting on the right side of the figure, a website widget is written using a query against the apparent canonical model. This query asks for the "SubjectName" from the canonical relation "Subject" as well as using our type operator (τ) to ask for the local attribute names for each returned tuple. The query uses the apparent model operator (κ) so that the query answer will be in the apparent model of the canonical structure. Following the green arrow to the left, the apparent model query then is translated into a query against the underlying model of the canonical structure. This query includes an id attribute that contains mapping and type provenance that will be used by the type operator in the first query. It then uses the canonical apply (θ) to retrieve results from all mapped domain structures via the translated query to the left following the blue arrow. The domain level query renames domain-level names to canonical-level names and uses the *apply* (α) operator to query the local databases. Based on the *apply* operator, the queries at the local level (left of the orange arrow) are generated. For each mapped local relation a query is generated by the *apply* operator that creates an id based on mapping provenance and renames local attributes to domain names. Results from mapped local relations (in the local databases) are unioned together.

The bottom section of Figure 3.5 shows how results from the local database are transformed into the apparent canonical form. Relational data from the local databases is renamed and nested into attributes containing provenance at the domain and canonical levels. That provenance is used at the canonical apparent level by the type operator[1]. A detailed explanation of this process is presented below

[1]As well as the operators introduced in the next chapter.

as we discuss the formalism and operation of each of the operators.

3.3 STRUCTURES AND MAPPINGS

In the following sections we define operators that make use of the nested relational formalism of local, domain, and canonical structures and the mappings between them that were defined in Chapter 2. We repeat these definitions in order to facilitate the understanding of the following sections.

Local DBs:

$ldb(\underline{id}, lrs(name, key, attrs(name)))$

A local database is defined by a tuple in the nested relation ldb. Each database is defined by an identifier id and a nested relation of the local relation names in the database. Each nested local relation tuple contains the name of a local relation, the name of the key attribute of the local relation, and a nested relation of the attribute names in that local relation. This nested relation stores the schema of all local databases.

Domain Structures:

$ds(\underline{id}, drs(name, key, attrs(name)))$

A domain structure is defined by a tuple in the nested relation ds. Each domain structure is defined by an identifier id and a nested relation of the domain relation names in the structure. Each nested domain relation tuple contains the name of a domain relation, the name of a key attribute, and a nested relation of the attribute names in that domain relation. This nested relation stores the schema of all domain structures.

Local DB - Domain Structure Mappings:

$ds\_ldb\_m(\underline{id}, ldbid, dsid, dr\_lr\_ms(id, lr, dr, p, corrs(id, la, da)))$

The set of all local-database-to-domain-structure mappings is defined in the ds_ldb_m nested relation. A mapping is defined by a tuple in this relation with an identifier id, the local database in the mapping $ldbid$, the domain structure in

the mapping $dsid$, and a nested relation of the relations mapped dr_lr_ms. Each relation mapping tuple in dr_lr_ms consists of an identifier id, the local relation lr, the domain relation dr, a predicate p, and a nested relation of the correspondences between local and domain attributes. Each correspondence consists of an identifier id, the local attribute name la, the domain attribute name da. The predicate in a mapping will be used in conditional mappings.

Canonical Structures:

$cs(\underline{id}, crs(name, key, attrs(name)))$

Each canonical structure is defined by a tuple in the nested relation cs with an identifier id and a nested relation of the canonical relation names in the structure. Each nested canonical-relation tuple contains the name of a canonical relation, the name of a key attribute, and a nested relation of the attribute names in that canonical relation. This nested relation contains the schema for all canonical structures.

Domain Structure - Canonical Structure Mappings:

$cs\_ds\_m(\underline{id}, dsid, csid, cr\_dr\_ms(id, dr, cr, corrs(id, da, ca)))$

The set of all domain-structure-to-canonical-structure mappings is defined in the cs_ds_m nested relation. A mapping is defined by a tuple in this relation with an identifier id, the domain structure in the mapping $dsid$, the canonical structure in the mapping $csid$, and a nested relation of the relations mapped cr_dr_ms. Each relation mapping tuple in cr_dr_ms consists of an identifier id, the domain relation dr, the canonical relation cr, and a nested relation of the correspondences between domain and canonical attributes. Each correspondence consists of an identifier id, the domain attribute name da, and the canonical attribute name ca. Unlike local-to-domain mappings, domain-to-canonical mappings do not allow predicates in the mappings.

We add two new definitions that tell our query interface which defined mappings will be relevant for a given query. We add this information for both the domain and canonical levels.

When a domain structure is used in an application or a widget, it is typically the case that only a subset of mappings will be used instead of all mappings that exist. In order to specify which mappings to use, we define domain structure applications.

Domain Structure Application:

$dsa(\underline{id}, dsid, ds\_ldb\_ms(ds\_ldb\_mid))$

The domain-structure-application relation consists of an identifier *id*, the domain structure *dsid*, and a nested relation of local-database-to-domain-structure mappings *ds_ldb_ms*, where each mapping is specified by its identifier *ds_ldb_mid*.

We then also define canonical structure applications so that we can specify which domain-structure-applications to use with a domain-structure-to-canonical-structure mapping.

Canonical Structure Application:

$csa(\underline{id}, csid, cs\_ds\_ms(dsaid, cs\_ds\_mid))$

We define the canonical-structure-application relation, which consists of an identifier *id*, the canonical structure *csid*, and a nested relation of domain-structure-to-canonical-structure mappings *cs_ds_ms*, where each mapping is specified by the domain structure application *dsaid* it uses and its identifier *cs_ds_mid*.

We show in depth examples below that use domain-structure-applications and canonical-structure-applications.

3.4 IMPLEMENTATION

Throughout this chapter we will show examples of our formalism as query results from an implementation of our formalism in a PostgreSQL database. We have implemented the above local-database, domain-structure, canonical-structure, and mapping definitions as nested relations within the database. While a full implementation of our query interface handles multiple local databases with a separate database for domain and canonical level structures, in this implementation we

simplify to a single database instance where each local database (with its data) is stored in its own namespace (what Postgres calls a "schema") and we store the definitions above in a separate namespace.

While Postgres is not a full nested-relational database, we use a combination of "row" attribute types and arrays to achieve equivalent functionality. Postgres supports the nest and unnest operations on these types, so we are able to fully translate our formalism into Postgres functions, which can then be used in queries against the canonical and domain levels.

In the following sections, examples will be shown using results from this implementation. Most examples start with a query and return results that are either part of the formalism or data from the local databases. All data from the implementation is labeled as a "Result Set", as shown above in Figures 3.2, 3.3, and 3.4.

3.5 APPLY

A query over a traditional database begins by choosing which relations to query and implicitly includes a *tablescan* operation to allow the rest of the query operators to address the result of that scan. We define the *apply* (α) operation that acts analogously to a *tablescan* for each domain relation in a query at the domain level; for example in Figure 3.5 we use the *apply* operator against $Person$ ($\alpha(Person)$) in the domain level query. The result of the *apply* operator to a domain relation is a query against mapped local databases that returns data in the format of the domain relation in the underlying model. Once the *apply* operator has been used on a domain relation, there will be a nested relational result in the underlying model that can be used with standard nested relational algebra operations—union, difference, projection, product, join, selection, rename—without any change in definition.

Given a domain structure application with id *dsaid* and a domain relation

named *dr*, the *apply* operator is defined as shown in Equation 3.1. We describe the operator in detail below through the use of a number of examples. The *apply* operator is designed to handle all cases allowed by our mapping system, from simple mappings to more complex mappings. For each of these cases, different parts of the formal definition come into play. In order to show how these different parts work, we present four examples. These examples are meant to demonstrate how all the various parts of the formalism work, they do not represent an exhaustive set of possible mappings in our system. After the examples, we will discuss how they can be combined to handle all possible mappings allowed by our system.

3.5.1 Simple Mappings

We begin with a set of simple mappings where each domain attribute is mapped a single time to a distinct local attribute for each local database. Figure 3.6 shows a subset of the tennis and football databases described above. Here they have been named "TennisSmallDB" and "FootballSmallDB" and each contains a single entity, "Student" and "Employee" respectively. The domain structure is a subset of the team domain structure named "ExampleDS" from Chapter 2 containing only the "Person" domain entity. In each mapping, the "PersonId" and "GivenName" domain attributes have been mapped to local attributes. This example shows how *apply* works in this straightforward case as well as showing how it deals with unmapped local attributes, which is to project them out of any result from the local database.

We will proceed through this example following the functions defined in Equation 3.1. To begin, let us look at the data within the system catalog for the local databases and domain structures in Figure 3.6.

Result Set 3.5

```
#select * from ldb;
       id        |                          lrs
-----------------+-------------------------------------------------
 FootballSmallDB | (Employee,EmployeeId,{EmployeeId,EmployeeName,Address})
 TennisSmallDB   | (Student,StudentId,{StudentId,StudentName})
```

$$\alpha(dr, dsaid) = \bigcup_{\substack{(ldbid, dr\_lr\_m) \in \\ \textbf{dsa\_mappings}(dr, dsaid)}} \Big(\textbf{mapped}(ldbid, dr\_lr\_m) \times \textbf{not\_mapped}(dr, dr\_lr\_m, dsaid) \Big) \tag{1}$$

$$\textbf{dsa\_mappings}(dr, dsaid) = \pi_{\substack{ds\_ldb\_m.ldbid, \\ ds\_ldb\_m.dr\_lr\_ms}} \left(\sigma_{\substack{ds\_ldb\_m.dr\_lr\_ms.dr = dr \\ \wedge dsa.id = dsaid}} \Big(ds\_ldb\_m \underset{ds\_ldb\_m.id = dsa.ds\_ldb\_mid}{\bowtie} (\mu_{ds\_ldb\_ms}(dsa)) \Big) \right) \tag{2}$$

$$\textbf{mapped}(ldbid, dr\_lr\_m) = \textbf{proj\_type\_nest}(da_1, ldbid, dr\_lr\_m) \underset{id}{\bowtie} \ldots \underset{id}{\bowtie} \textbf{proj\_type\_nest}(da_n, ldbid, dr\_lr\_m) \tag{3}$$
$$\forall da_i \in \pi_{dr\_lr\_m.corrs.da}(dr\_lr\_m)$$

$$\textbf{proj\_type\_nest} \quad (da, ldbid, dr\_lr\_m) = \nu_{value, meta:[da]} \left(\bigcup_{corr \in \pi_{dr\_lr\_m.corrs}(\sigma_{dr\_lr\_m.corrs.da = da}(dr\_lr\_m))} \Big(\tag{4}$$
$$\big(\pi_{\substack{\textbf{gen\_key}(ldbid, dr\_lr\_m) \rightarrow id, \\ [corr.la] \rightarrow value}} (\sigma_{dr\_lr\_m.p}(\textbf{table\_scan}(ldbid, dr\_lr\_m.lr))) \big) \times \underset{\rightarrow meta(mid, cid, type)}{(dr\_lr\_m.id, corr.id, corr.la)} \Big) \Big)$$

$$\textbf{gen\_key}(ldbid, dr\_lr\_m) = ldbid||.||dr\_lr\_m.lr||.||dr\_lr\_m.id||.||[\pi_{ldb.lrs.key}(\sigma_{\substack{ldb.lrs.name = dr\_lr\_m.lr \\ \wedge ldb.id = ldbid}}(ldb)] \tag{5}$$

$$\textbf{not\_mapped}(dr, dr\_lr\_m, dsaid) = \bigtimes_{\substack{da \in \textbf{dattrs}(dr, dsaid) \wedge \\ \textbf{is\_empty}(\sigma_{dr\_lr\_m.corrs.da = [da]}(dr\_lr\_m))}} \underset{\rightarrow [da](value, meta(mid, cid, type))}{(NULL, (dr\_lr\_m.id, NULL, NULL))} \tag{6}$$

$$\textbf{dattrs}(dr, dsaid) = \pi_{ds.drs.attrs}(\sigma_{\substack{ds.drs.name = [dr] \\ \wedge dsa.id = [dsaid]}}(ds \underset{ds.id = dsa.dsid}{\bowtie} dsa)) \tag{7}$$

Where the **table_scan** $(ldbid, lr)$ function performs a table scan operation on the local relation lr in the local database $ldbid$ and the **is_empty** $(query)$ function returns a boolean: true if the input query returns no rows, false otherwise.

Equation 3.1: Apply

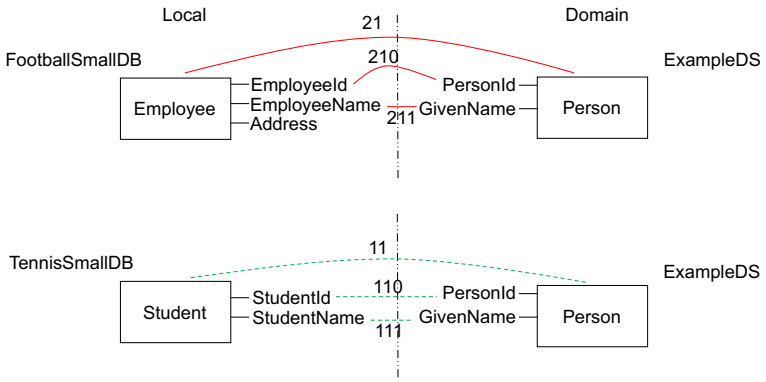


Figure 3.6: Two straightforward mappings. Above, a mapping between the "Employee" local relation and the "Person" domain relation with correspondences between "EmployeeId" and "PersonId", and "EmployeeName" and "GivenName". Below, a mapping between the "Student" local relation and the "Person" domain relation with correspondences between "StudentId" and "PersonId", and "StudentName" and "GivenName"

As described above and shown in Figure 3.6, we have two local databases, "FootballSmallDB" and "TennisSmallDB", each with a single relation, "Employee" and "Student"; each relation has a key attribute, "EmployeeId" and "StudentId". The "Employee" relation has three attributes, "EmployeeId", "EmployeeName", and "Address"; and the "Student" relation has two attributes, "StudentId" and "StudentName".

Result Set 3.6

```
# select * from ds;
    id     |                drs
-----------+-----------------------------------------
 ExampleDS | (Person,PersonId,{PersonId,GivenName})
```

The domain structure is named "ExampleDS" and has a single domain relation, "Person", which has attributes "PersonId" (which is the key attribute) and "GivenName".

For the sake of these examples each database has a small amount of data shown below.

Result Set 3.7

```
# select * from student;
 studentid |  studentname
-----------+---------------
 800       | Callie Reese
 801       | Kibo Nolan
 802       | Aiko Sweet
 803       | Elton Duncan
 804       | Macaulay Hess
```

Result Set 3.8

```
# select * from employee
 employeeid | employeename  |              address
------------+---------------+-----------------------------------
 9760       | Raja Ryan     | 882-7477 Neque St.
 4187       | Mary Stone    | P.O. Box 903, 4348 Eget St.
 7040       | Amelia Little | P.O. Box 399, 2901 Ut Avenue
 5271       | Hasad Wagner  | 8767 Faucibus St.
 1578       | Dylan Miles   | P.O. Box 194, 2522 Facilisis St.
```

The mappings shown in Figure 3.6 are represented in the implementation as follows:

Result Set 3.9

```
# select * from ds_ldb_m;
 id |      ldbid      |   dsid    |                          dr_lr_ms
----+-----------------+-----------+-------------------------------------------------------------
 1  | TennisSmallDB   | ExampleDS | (11,Student,Person,TRUE,{(110,StudentId,PersonId),
                                                          (111,StudentName,GivenName)})
 2  | FootballSmallDB | ExampleDS | (21,Employee,Person,TRUE,{(210,EmployeeId,PersonId),
                                                           (211,EmployeeName,GivenName)})
```

The red mapping in the top of Figure 3.6 is shown in the second tuple with id “2”. This mapping is between the local database “FootballSmallDB” and the domain structure “ExampleDS”. It consists of mapping “21” between the “Employee” local relation and the “Person” domain relation, which contains correspondences “210” (between the “EmployeeId” local attribute and the “PersonId” domain attribute) and “211” (between the “EmployeeId” local attribute and the “PersonId” domain attribute). The predicate for the mapping is “TRUE”, the default predicate, which means that all local tuples will be passed to the domain level based on these correspondences.

The green mapping in the bottom of Figure 3.6 is shown in the first tuple with id “1” between the local database “TennisSmallDB” and the “ExampleDS” domain structure.

For this example we have a domain-structure-application that uses these two mappings, shown below with id “3”.

Result Set 3.10

```
# select * from dsa;
 id |   dsid    | ds_ldb_mid
----+-----------+------------
 3  | ExampleDS | {1,2}
```

In order to produce the output of the *apply* operator on the "Person" domain structure using the domain structure application with id "3" above, $\alpha(Person, 3)$, we will show the various steps of the formalism using the Postgres implementation. Recall from Function 3.1.1 that we need to first determine which mappings will be used in the union using the **dsa_mappings** function (Function 3.1.2). Note, for clarity, the functions in the implementation have been modified from the formalism to pass and return the ids of mappings instead of full mappings.

Result Set 3.11

```
# select * from dsa_mappings_id('Person','3');
       ldb       | mid
-----------------+-----
 TennisSmallDB   | 11
 FootballSmallDB | 21
```

We see that the **dsa_mappings** function returns the two local-relation-to-domain-relation mappings specified in the two local-database-to-domain-structure mappings in the "dsa" relation above. For each of these two mappings, we explain the *apply* operation by examining what happens in each of its functions. For the first mapping, with id "11", we start by looking into the **mapped** function, Function 3.1.3. The **mapped** function combines the output of the **proj_type_nest** function for each domain attribute in the specified mappings. We see from above that there are two attributes in mapping "11", "PersonId" and "GivenName". The result of **proj_type_nest** for the "PersonId" domain attribute is shown below.

Result Set 3.12

```
# select * from projtypenest_id('PersonId','TennisSmallDB','11');
              id               | personid(value,meta(mid,cid,type))
-------------------------------+-----------------------------------
 TennisSmallDB.Student.11.800 | {(800,(11,110,StudentId))}
 TennisSmallDB.Student.11.802 | {(802,(11,110,StudentId))}
 TennisSmallDB.Student.11.801 | {(801,(11,110,StudentId))}
 TennisSmallDB.Student.11.803 | {(803,(11,110,StudentId))}
 TennisSmallDB.Student.11.804 | {(804,(11,110,StudentId))}
```

The first attribute in the output above is the "id" attribute. This attribute is generated by the **gen_key** function (Function 3.1.5) by combining the local database

("TennisSmallDB"), the local relation ("Student"), the mapping id ("11"), and the key from the local database (in this case the "StudentId"). The second nested attribute is created by the second and third lines of the **proj_type_nest** function (Function 3.1.4). The "value" nested attribute is created by the **table_scan** on the local database based on correspondence "110" in mapping "11" (in this case the values "800", ..., "804" come from the "StudentId" attribute from the "Student" local relation as shown in Result Set 3.7). This correspondence contains the local attribute "StudentId" and the mapping has the predicate "TRUE", so on line two of the **proj_type_nest** function (Function 3.1.4) the relational select will select all tuples from the local database and the project operator will project out the "StudentId" attribute and rename it to "value". This renaming is done in the $[corr.la] \rightarrow value$ argument to the project operator. Note, we use the square brackets to reference the dynamic variable *corr.la*, which is populated from the correspondence from the input mapping and we use the right arrow to indicate the renaming syntax within the project operator. Lastly, we create the nested "meta" attribute in line three of the **proj_type_nest** function (Function 3.1.4) with the mapping id, "11", the correspondence id, "110", and the local attribute name, "StudentId". These three attributes are nested in the "meta" attribute with the *attribute naming* operator ($\rightarrow$). The "meta" attribute is then cross joined with the rest of the output and then nested with the "value" attribute using the nest operator, ν, on line one of the **proj_type_nest** function (Function 3.1.4). The "value" and "meta" attributes are nested into a new attribute called "PersonId" from the domain attribute of the correspondence. Once again we use the square bracket notation to denote that *da* is dynamically populated from the correspondence.

As we see in the **mapped** function (3.1.3), we will perform the **proj_type_nest** function for each attribute in the mapping returned from the **dsa_mappings** function. In this case, for the "TennisSmallDB" database, we perform the **proj_type_nest** function for the "PersonId" attribute shown above and the

"GivenName" attribute shown below.

Result Set 3.13

```
# select * from projtypenest_id('GivenName','TennisSmallDB','11');
              id              | givenname(value,meta(mid,cid,type))
------------------------------+-----------------------------------------------------
 TennisSmallDB.Student.11.800 | {(Callie Reese,(11,111,StudentName))}
 TennisSmallDB.Student.11.802 | {(Aiko Sweet,(11,111,StudentName))}
 TennisSmallDB.Student.11.801 | {(Kibo Nolan,(11,111,StudentName))}
 TennisSmallDB.Student.11.803 | {(Elton Duncan,(11,111,StudentName))}
 TennisSmallDB.Student.11.804 | {(Macaulay Hess,(11,111,StudentName))}
```

As with the "PersonId" attribute, we first generate the "id" attribute and then we create a nested attribute for "GivenName". In this case, the local attribute in correspondence "111" is "StudentName", so we retrieve the local data for "StudentName" and then create the "meta" attribute accordingly.

Once the results of the **proj_type_nest** function for each of the mapped attributes of mapping "11" have been returned, the **mapped** function (3.1.3) joins the result of each **proj_type_nest** function using the generated "id" attribute to recombine the local data back into tuples similar in structure to the original local database tuples[2].

Result Set 3.14

```
# select * from mapped_id('ExampleDS','Person','TennisSmallDB','11');
              id              |              personid              |                 givenname
------------------------------+------------------------------------+-----------------------------------
 TennisSmallDB.Student.11.800 |{(800,(11,110,StudentId))} |{(Callie Reese,(11,111,StudentName))}
 TennisSmallDB.Student.11.802 |{(802,(11,110,StudentId))} |{(Aiko Sweet,(11,111,StudentName))}
 TennisSmallDB.Student.11.801 |{(801,(11,110,StudentId))} |{(Kibo Nolan,(11,111,StudentName))}
 TennisSmallDB.Student.11.803 |{(803,(11,110,StudentId))} |{(Elton Duncan,(11,111,StudentName))}
 TennisSmallDB.Student.11.804 |{(804,(11,110,StudentId))} |{(Macaulay Hess,(11,111,StudentName))}
```

With the results above, we now have the output from the *apply* operation on the "Person" domain relation for a single local database ("TennisDBSmall"). We next repeat the steps above for all remaining mappings returned from the **dsa_mappings** function shown in Result Set 3.11, in this case mapping "21", the green lines shown in Figure 3.6 and the second tuple in Result Set 3.9. Like the first mapping described, this mapping also contains correspondences to the "PersonId" and "GivenName" domain attributes, this time from the "EmployeeId"

[2]In the coming examples below we discuss why we break apart and reassemble the original tuples. Also, later in this chapter we will discuss optimizations in the cases where we can avoid doing so.

and “EmployeeName” local attributes in the “Employee” local relation in the “FootballSmallDB” local database. For this mapping we then repeat the **mapped** function which starts by running the **proj_type_nest** function for “PersonId” shown below.

Result Set 3.15

```
# select * from projtypenest_id('PersonId','FootballSmallDB','21');
                id                | personid(value,meta(mid,cid,type))
----------------------------------+-----------------------------------
 FootballSmallDB.Employee.21.4187 | {(4187,(21,210,EmployeeId))}
 FootballSmallDB.Employee.21.7040 | {(7040,(21,210,EmployeeId))}
 FootballSmallDB.Employee.21.5271 | {(5271,(21,210,EmployeeId))}
 FootballSmallDB.Employee.21.9760 | {(9760,(21,210,EmployeeId))}
 FootballSmallDB.Employee.21.1578 | {(1578,(21,210,EmployeeId))}
```

As before, the “id” attribute is generated with the local database name, the local relation name, the mapping id, and then the key attribute from the local database; in this case “FootballSmallDB.Employee.21” and the local key data which is from the “EmployeeId” local attribute. Recall that this key attribute is shown in the first tuple in Result Set 3.5. Then, following the **proj_type_nest** function (Function 3.1.4), we project the “EmployeeId” attribute from the “Employee” relation, rename it to “value”, and then nest it with the generated “meta” attribute into the “PersonId” attribute. We repeat the **proj_type_nest** function for the next mapped attribute, “GivenName”.

Result Set 3.16

```
# select * from projtypenest_id('GivenName','FootballSmallDB','21');
                id                | givenname(value,meta(mid,cid,type))
----------------------------------+-----------------------------------------------------
 FootballSmallDB.Employee.21.4187 | {(Mary Stone,(21,211,EmployeeName))}
 FootballSmallDB.Employee.21.7040 | {(Amelia Little,(21,211,EmployeeName))}
 FootballSmallDB.Employee.21.5271 | {(Hasad Wagner,(21,211,EmployeeName))}
 FootballSmallDB.Employee.21.9760 | {(Raja Ryan,(21,211,EmployeeName))}
 FootballSmallDB.Employee.21.1578 | {(Dylan Miles,(21,211,EmployeeName))}
```

As with the “PersonId” attribute, we first generate the “id” attribute and then we create a nested attribute for “GivenName”. In this case the local attribute in correspondence “211” is “EmployeeName”, so we retrieve the local data for “EmployeeName” and then create the “meta” attribute accordingly.

As above, the returned results from the **proj_type_nest** functions are joined on the generated “id” attribute and the results of the **mapped** function are shown below[3].

[3]Note that attribute values have been shortened to fit the page using “...”, the attributes

Result Set 3.17

```
# select * from mapped_id('ExampleDS','Person','FootballSmallDB','21');
                id                |            personid            |            givenname
----------------------------------+--------------------------------+---------------------------------
FootballSmallDB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.7040|{(7040,(21,210,EmployeeId))}|{(Amelia...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.5271|{(5271,(21,210,EmployeeId))}|{(Hasad...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.1578|{(1578,(21,210,EmployeeId))}|{(Dylan...,(21,211,EmployeeName))}
```

We now have the results of all mapped domain attributes from all the local databases. This examples also shows what happens to local attributes that are not included in mappings, in this case the "Address" attribute from the "FootballSmallDB" was not included in mapping "21". Each time the **proj_type_nest** function ran, the "Address" attribute was not included in the project list and therefore not included in any results[4].

With the results from the **mapped** function for all the mappings returned, the last step in the *apply* operator (α) (applied to the "Person" domain relation for this example) is to union the results as shown below.

Result Set 3.18

```
# select * from alpha_id('Person','3');
                id                |              personid          |             givenname
----------------------------------+--------------------------------+-----------------------------------
TennisSmallDB.Student.11.800    |{(800,(11,110,StudentId))}  |{(Callie ...,(11,111,StudentName))}
TennisSmallDB.Student.11.802    |{(802,(11,110,StudentId))}  |{(Aiko ...,(11,111,StudentName))}
TennisSmallDB.Student.11.801    |{(801,(11,110,StudentId))}  |{(Kibo ...,(11,111,StudentName))}
TennisSmallDB.Student.11.803    |{(803,(11,110,StudentId))}  |{(Elton ...,(11,111,StudentName))}
TennisSmallDB.Student.11.804    |{(804,(11,110,StudentId))}  |{(Macaulay...,(11,111,StudentName))}
FootballSmallDB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.7040|{(7040,(21,210,EmployeeId))}|{(Amelia...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.5271|{(5271,(21,210,EmployeeId))}|{(Hasad...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.1578|{(1578,(21,210,EmployeeId))}|{(Dylan...,(21,211,EmployeeName))}
```

In this example, all domain attributes in the domain relation have been mapped in each mapping, so the **not_mapped** function (Function 3.1.6) was not used. We will discuss its functionality next.

---

contain the data shown in Results Sets 3.15 and 3.16. This convention will be used throughout this chapter.

[4]We will show in the next chapter how this local data can still be accessed from the domain and canonical levels.

3.5.2 Unmapped Domain Attributes

In this example, we explain what happens when there are domain attributes for which there are no correspondences to a local database in a mapping. This example will demonstrate the **not_mapped** function (3.1.6), the only function not used in the previous example. Figure 3.7 closely resembles the schema and mappings shown in Figure 3.6; the difference is that the domain relation "Person" now has a third attribute "PersonAddress". A correspondence ("212") has been added between the "Address" attribute in the "Employee" local relation and the "PersonAddress" domain attribute. The "Student" local relation has no applicable attribute, in this case the "PersonAddress" domain attribute is unmapped.

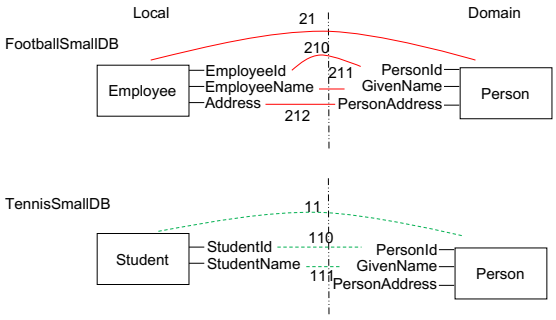


Figure 3.7: Above, a mapping between domain and local where all domain attributes are mapped. Below, a mapping where the "PersonAddress" domain attribute has no correspondences with any local attribute.

For this example the local databases and local data remained unchanged from those used in the previous example shown in Result Sets 3.5, 3.7, and 3.8. The domain structure is updated by adding the "PersonAddress" domain attribute shown below in Result Set 3.19 highlighted in red.

Result Set 3.19

```
# select * from ds;
    id     |                drs
-----------+-----------------------------------------
 ExampleDS | (Person,PersonId,{PersonId,GivenName,PersonAddress})
```

The local-database-to-domain-structure mappings shown in Result Set 3.9 are

updated by adding the new correspondence ("212") between the "Address" attribute in the "Employee" local relation and the "PersonAddress" attribute in the "Person" domain relation. The new correspondence is reflected below in Result Set 3.20, highlighted in red.

Result Set 3.20

```
# select * from ds_ldb_m;
 id |      ldbid      |   dsid    |      dr_lr_ms
----+-----------------+-----------+-----------------------------------------------------------
  2 | FootballSmallDB | ExampleDS | (21,Employee,Person,TRUE,{(210,EmployeeId,PersonId),
                                                              (211,EmployeeName,GivenName),
                                                              (212,Address,PersonAddress)})
  1 | TennisSmallDB   | ExampleDS | (11,Student,Person,TRUE,{(110,StudentId,PersonId),
                                                             (111,StudentName,GivenName)})
```

Since the local-database-to-domain-structure mapping ids and local-relation-to-domain-relation mapping ids have remained the same, we use the same domain-structure-application shown in Result Set 3.10. Then the output of the *apply* operator on the "Person" domain relation using domain structure application with id "3", $\alpha(Person, 3)$, will be produced with the following steps. Since the mapping ids have not changed from our previous example the result from the **dsa_mappings** function (Function 3.1.2) will be the same as that shown in Result Set 3.11.

Starting with the mapping from the "Employee" local relation ("21") where all domain attributes are mapped, we proceed as in the example above. The **mapped** function (3.1.3) will combine the output from the **proj_type_nest** function (3.1.4) for each mapped domain attribute. The results for "PersonId" and "GivenName" were shown previously in Result Sets 3.15 and 3.16. We combine those outputs with the output of the **proj_type_nest** function for the new correspondence to the "PersonAddress" attribute, shown below in Result Set 3.21.

Result Set 3.21

```
# select * from projtypenest_id('PersonAddress','FootballSmallDB','22');
                id                |      personaddress(value,meta(mid,cid,type))
----------------------------------+-----------------------------------------------------------
FootballSmallDB.Employee.22.7040 | {(P.O. Box 399, 2901 Ut Avenue,(22,221,Address))}
FootballSmallDB.Employee.22.9760 | {(882-7477 Neque St.,(22,221,Address))}
FootballSmallDB.Employee.22.5271 | {(8767 Faucibus St.,(22,221,Address))}
FootballSmallDB.Employee.22.1578 | {(P.O. Box 194, 2522 Facilisis St.,(22,221,Address))}
FootballSmallDB.Employee.22.4187 | {(P.O. Box 903, 4348 Eget St.,(22,221,Address))}
```

These results are then combined in the **mapped** function (3.1.3) as follows in Result Set 3.22.

Result Set 3.22

```
# select * from mapped_id('ExampleDS','Person','FootballSmallDB','22');
              id       |          personid          |givenname|        personaddress
-----------------------+----------------------------+---------+-----------------------------------
Foot...DB.Emp...22.7040|{(7040,(22,222,Emp...Id))}|{(Amel...|{(P.O. Box 399...,(22,221,Address))}
Foot...DB.Emp...22.9760|{(9760,(22,222,Emp...Id))}|{(Raja...|{(882-7477 Neq...,(22,221,Address))}
Foot...DB.Emp...22.5271|{(5271,(22,222,Emp...Id))}|{(Hasa...|{(8767 Faucibu...,(22,221,Address))}
Foot...DB.Emp...22.1578|{(1578,(22,222,Emp...Id))}|{(Dyla...|{(P.O. Box 194...,(22,221,Address))}
Foot...DB.Emp...22.4187|{(4187,(22,222,Emp...Id))}|{(Mary...|{(P.O. Box 903...,(22,221,Address))}
```

Thus far we have shown only mapped attributes. We now present how the **not_mapped** function (3.1.6) works in the case of the "Student" local relation, which does not have any attributes that correspond to the "PersonAddress" domain attribute in mapping "11". To begin, we perform the **mapped** function (3.1.3) for the attributes that do have correspondences. For "PersonId" and "GivenName" domain attributes the **mapped** function invokes the **proj_type_nest** function (3.1.4) as in Result Sets 3.12 and 3.13, that produces the same output for the **mapped** function as Result Set 3.14. Which leaves the "PersonAddress" domain attribute, which does not appear in any correspondences in mapping "11", so will produce a result from the **not_mapped** function (3.1.6). The **not_mapped** function will produce a single tuple as its output that contains all domain attributes that were not mapped. We get all domain attributes in the relation using the $da \in$ **dattrs** line in the cross product and then check each domain attribute to see if it was not mapped by making sure that there are no correspondences containing the domain attribute using the **is_empty** function. Having determined in this case that the "PersonAddress" attribute does not have a correspondence, the function produces a single tuple with a single nested attribute shown below in Result Set 3.23.

Result Set 3.23

```
# select * from not_mapped_id('ExampleDS','Person','TennisSmallDB','11');
   personaddress(value,meta(mid,cid,type))
-------------------------------------------
  {(NULL,(11,NULL,NULL))}
```

This result corresponds to the right side of the **not_mapped** function (3.1.6). A tuple is produced with "NULL" values for the "value", "cid", and "type" nested

attributes, since there were no corresponding mappings or local data or type information. The attribute is named using our naming operator (→) to produce a result using the schema of a domain attribute. This result is then crossed with the result of the **mapped** function as per the first line of the *apply* operator (3.1.1) as shown in Result Set 3.24.

Result Set 3.24

```
# select * from mapped_id('ExampleDS','Person','TennisSmallDB','11'),
              not_mapped_id('ExampleDS','Person','TennisSmallDB','11');
              id               |             personid        | givenname | personaddress
-------------------------------+-----------------------------+-----------+---------------
 TennisSmallDB.Student.11.800 | {(800,(11,110,StudentId))} | {(Call... | {(NULL,(11,NULL,NULL))}
 TennisSmallDB.Student.11.802 | {(802,(11,110,StudentId))} | {(Aiko... | {(NULL,(11,NULL,NULL))}
 TennisSmallDB.Student.11.801 | {(801,(11,110,StudentId))} | {(Kibo... | {(NULL,(11,NULL,NULL))}
 TennisSmallDB.Student.11.803 | {(803,(11,110,StudentId))} | {(Elto... | {(NULL,(11,NULL,NULL))}
 TennisSmallDB.Student.11.804 | {(804,(11,110,StudentId))} | {(Maca... | {(NULL,(11,NULL,NULL))}
```

Results Sets 3.22 and 3.24 are then unioned to produce the result of the apply operator, shown below in Result Set 3.25.

Result Set 3.25

```
# select * from alpha_id('Person','3');
                id                |            personid          |    givenname    |                    personaddress
----------------------------------+------------------------------+-----------------+-----------------------------------------------------
Tennis...DB.Student.11.800 |{(800,(11,110,StudentId))}  |{(Callie Rees...}| {(NULL,(11,NULL,NULL))}
Tennis...DB.Student.11.802 |{(802,(11,110,StudentId))}  |{(Aiko Sweet,...}| {(NULL,(11,NULL,NULL))}
Tennis...DB.Student.11.801 |{(801,(11,110,StudentId))}  |{(Kibo Nolan,...}| {(NULL,(11,NULL,NULL))}
Tennis...DB.Student.11.803 |{(803,(11,110,StudentId))}  |{(Elton Dunca...}| {(NULL,(11,NULL,NULL))}
Tennis...DB.Student.11.804 |{(804,(11,110,StudentId))}  |{(Macaulay He...}| {(NULL,(11,NULL,NULL))}
Foot...DB.Employee.22.7040 |{(7040,(22,222,EmployeeId))}|{(Amelia Litt...}| {(P.O. Box 399, 2901 Ut Avenue,(22,221,Address))}
Foot...DB.Employee.22.9760 |{(9760,(22,222,EmployeeId))}|{(Raja Ryan,(...}| {(882-7477 Neque St.,(22,221,Address))}
Foot...DB.Employee.22.5271 |{(5271,(22,222,EmployeeId))}|{(Hasad Wagne...}| {(8767 Faucibus St.,(22,221,Address))}
Foot...DB.Employee.22.1578 |{(1578,(22,222,EmployeeId))}|{(Dylan Miles...}| {(P.O. Box 194, 2522 Facilisis St.,(22,221,Address))}
Foot...DB.Employee.22.4187 |{(4187,(22,222,EmployeeId))}|{(Mary Stone,...}| {(P.O. Box 903, 4348 Eget St.,(22,221,Address))}
```

3.5.3 Multiple Local Attributes Mapped to One Domain Attribute

In this example we explain what happens when there are multiple correspondences from different local attributes to the same domain attribute in a single mapping. This example will demonstrate why we use the nested relational model for each of our attributes. Figure 3.8 shows an example of this case. In this example, we extend the domain relation “Person” from the simple case (Figure 3.2) with a new domain attribute “OrganizationalUnit”. The “Employee” local relation is extended with both “Division” and “Region” attributes, which represent the league and the geographical region within which the team plays, respectively. Correspondences “214” and “213” have been added between “Division” and “OrganizationalUnit” and “Region” and “Organizational Unit”, respectively. The “Student” local relation is extended with a “Department” attribute, which represents the academic

department within which the student studies and the correspondence “112” between “Department” and OrganizationalUnit”.

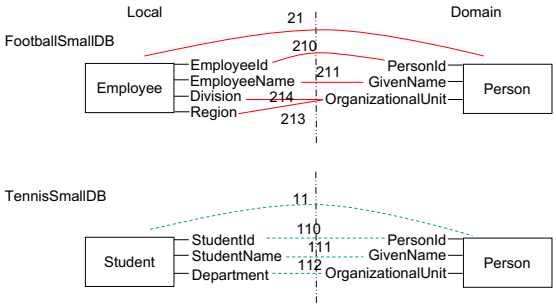


Figure 3.8: Above, an example of a mapping with two correspondences containing the same domain attribute (“OrganizationalUnit”). Below, a straightforward mapping.

The local database definition is updated to reflect the new local attributes as shown below in Result Set 3.26 in red.

Result Set 3.26

```
select * from ldb;
       id        |                                                  lrs
-----------------+-------------------------------------------------------------------
 FootballSmallDB | (Employee,EmployeeId,{EmployeeId,EmployeeName,Division,Region})
 TennisSmallDB   | (Student,StudentId,{StudentId,StudentName,Department})
```

The domain-structure definition is updated to reflect the new domain attribute as shown below in Result Set 3.27 in red.

Result Set 3.27

```
# select * from ds;
    id     |                          drs
-----------+-------------------------------------------------------------
 ExampleDS | (Person,PersonId,{PersonId,GivenName,OrganizationalUnit})
```

Sample data for the two local relations is shown below in Result Sets 3.28 and 3.29.

Result Set 3.28

```
# select * from student;
 studentid |  studentname  | department
-----------+---------------+-------------
 800       | Callie Reese  | physics
 801       | Kibo Nolan    | math
 802       | Aiko Sweet    | english
 803       | Elton Duncan  | french
 804       | Macaulay Hess | biology
```

Result Set 3.29

```
# select * from employee;
 employeeid | employeename  | division | region
------------+---------------+----------+---------
 9760       | Raja Ryan     | league 1 | europe
 4187       | Mary Stone    | league 2 | america
 7040       | Amelia Little | league 2 | america
 5271       | Hasad Wagner  | league 1 | europe
 1578       | Dylan Miles   | league 2 | america
```

As in the previous examples, the new correspondences are added to the mappings, shown in red in Result Set 3.30. For the football database, the first new correspondence is between the "Region" local attribute and the "OrganizationalUnit" domain attribute with id "213" and the second is between the "Division" local attribute and the "OrganizationalUnit" domain attribute with id "214". For the student database, we add the new correspondence between the "Department" local attribute and the "OrganizationalUnit" domain attribute with id "112".

Result Set 3.30

```
 # select * from ds_ldb_m;
 id |      ldbid      |   dsid    |                                dr_lr_ms
----+-----------------+-----------+-------------------------------------------------------------------------
  2 | FootballSmallDB | ExampleDS |(21,Employee,Person,TRUE,{(210,EmployeeId,PersonId),
                                                           (211,EmployeeName,GivenName),
                                                           (213,Region,OrganizationalUnit),
                                                           (214,Division,OrganizationalUnit)})
  1 | TennisSmallDB   | ExampleDS |(11,Student,Person,TRUE,{(110,StudentId,PersonId),
                                                           (111,StudentName,GivenName),
                                                           (112,Department,OrganizationalUnit)})
```

Since the local-database-to-domain-structure mapping ids and local-relation-to-domain-relation mapping ids have remained the same, we use the same domain-structure-application shown in Result Set 3.10. Since the "TennisSmallDB" database remains a case of simple mappings we do not repeat the description of the steps to produce the results of the mapped function, shown below in Result Set 3.31.

Result Set 3.31

```
#select * from mapped_id('ExampleDS','Person','TennisSmallDB','11');
              id              |            personid            |givenname|      organizationalunit
------------------------------+--------------------------------+---------+------------------------------
TennisSmallDB.Student.11.800|{(800,(11,110,StudentId))}|{(Call...|{(physics,(11,112,Department))}
TennisSmallDB.Student.11.801|{(801,(11,110,StudentId))}|{(Kibo...|{(math,(11,112,Department))}
TennisSmallDB.Student.11.802|{(802,(11,110,StudentId))}|{(Aiko...|{(english,(11,112,Department))}
TennisSmallDB.Student.11.804|{(804,(11,110,StudentId))}|{(Maca...|{(biology,(11,112,Department))}
TennisSmallDB.Student.11.803|{(803,(11,110,StudentId))}|{(Elto...|{(french,(11,112,Department))}
```

The portion of interest in this example occurs in the **proj_type_nest** function (Function 3.1.4 in Equation 3.1) for the "FootballSmallDB". Without the nest operator at the beginning, the function would produce tuples of the form $(id, value, meta)$ where "value" is the local data value and "meta" is the nested provenance and type information. In this case, where multiple local attributes ("Region" and "Division") have been mapped to the same domain attribute ("Organizational Unit"), the union operator over all correspondences for this domain

attribute will then produce multiple tuples for each id. The nest operator then causes the "value" and "meta" attributes to be nested into a new attribute with the name of the domain attribute ([*da*]) where we use the bracket operators to signify that this is the value of the *da* parameter passed into the function. The nested result from the football database is shown below in Result Set 3.32.

Result Set 3.32

```
# select * from projtypenest_id('OrganizationalUnit','FootballSmallDB','21');
                id                |  organizationalunit(value,meta(mid,cid,type))
----------------------------------+-----------------------------------------------------------------
 FootballSmallDB.Employee.21.5271 | {(europe,(21,213,Region)),(league 1,(21,214,Division))}
 FootballSmallDB.Employee.21.9760 | {(europe,(21,213,Region)),(league 1,(21,214,Division))}
 FootballSmallDB.Employee.21.7040 | {(america,(21,213,Region)),(league 2,(21,214,Division))}
 FootballSmallDB.Employee.21.4187 | {(america,(21,213,Region)),(league 2,(21,214,Division))}
 FootballSmallDB.Employee.21.1578 | {(america,(21,213,Region)),(league 2,(21,214,Division))}
```

The nested result from the **proj_type_nest** function is then joined into the **mapped** and **apply** functions and then unioned together like usual, shown below in Result Sets 3.33 and 3.34.

Result Set 3.33

```
# select * from mapped_id('ExampleDS','Person','FootballSmallDB','21');
          id             |          personid           |givenname|  organizationalunit
-------------------------+-----------------------------+---------+-----------------------------
Foot...DB.Employee.21.5271|{(5271,(21,210,EmployeeId))}|{(Hasa...|{(europe,(21,213,Region)),
                                                                    (league 1,(21,214,Division))}
Foot...DB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...|{(europe,(21,213,Region)),
                                                                    (league 1,(21,214,Division))}
Foot...DB.Employee.21.7040|{(7040,(21,210,EmployeeId))}|{(Amel...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
Foot...DB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
Foot...DB.Employee.21.1578|{(1578,(21,210,EmployeeId))}|{(Dyla...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
```

Result Set 3.34

```
# select * from alpha_id('Person','3');
            id            |           personid          |givenname|     organizationalunit
--------------------------+-----------------------------+---------+-----------------------------
Tennis...DB.Student.11.800|{(800,(11,110,StudentId))}  |{(Call...|{(physics,(11,112,Department))}
Tennis...DB.Student.11.801|{(801,(11,110,StudentId))}  |{(Kibo...|{(math,(11,112,Department))}
Tennis...DB.Student.11.802|{(802,(11,110,StudentId))}  |{(Aiko...|{(english,(11,112,Department))}
Tennis...DB.Student.11.804|{(804,(11,110,StudentId))}  |{(Maca...|{(biology,(11,112,Department))}
Tennis...DB.Student.11.803|{(803,(11,110,StudentId))}  |{(Elto...|{(french,(11,112,Department))}
Foot...DB.Employee.21.5271|{(5271,(21,210,EmployeeId))}|{(Hasa...|{(europe,(21,213,Region)),
                                                                    (league 1,(21,214,Division))}
Foot...DB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...|{(europe,(21,213,Region)),
                                                                    (league 1,(21,214,Division))}
Foot...DB.Employee.21.7040|{(7040,(21,210,EmployeeId))}|{(Amel...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
Foot...DB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
Foot...DB.Employee.21.1578|{(1578,(21,210,EmployeeId))}|{(Dyla...|{(america,(21,213,Region)),
                                                                    (league 2,(21,214,Division))}
```

3.5.4 Conditional Mapping Predicates

In this example, we present a use case that demonstrates why the predicate exists within the local-to-domain mappings. For this example, we extend the simple case

(Figure 3.2) by adding attributes to the "Student" and "Employee" local relations to include gender, "Sex" and "Gender" respectively. These attributes are shown in Result Set 3.35 in red.

Result Set 3.35

```
# select * from ldb;
       id        |                        lrs
-----------------+-----------------------------------------------------------
 FootballSmallDB | (Employee,EmployeeId,{EmployeeId,EmployeeName,Gender})
 TennisSmallDB   | (Student,StudentId,{StudentId,StudentName,Sex})
```

Sample local data for the two local relations is shown below in Result Sets 3.36 and 3.37.

Result Set 3.36

```
 studentid |  studentname  |  sex
-----------+---------------+--------
 800       | Callie Reese  | female
 801       | Kibo Nolan    | male
 802       | Aiko Sweet    | female
 803       | Elton Duncan  | male
 804       | Macaulay Hess | male
```

Result Set 3.37

```
 employeeid | employeename  | gender
------------+---------------+--------
 9760       | Raja Ryan     | f
 4187       | Mary Stone    | f
 7040       | Amelia Little | f
 5271       | Hasad Wagner  | m
 1578       | Dylan Miles   | m
```

For this use case, we modify the domain structure where we replace the person domain relation with two new domain relations "Female" and "Male" shown in Result Set 3.38. Each domain relation has attributes for id and name, "FemaleId" and "FName" for the "Female" domain relation; and, "MaleId" and "MName" for the "Male" domain relation.

Result Set 3.38

```
# select * from ds;
    id     |            drs
-----------+--------------------------------
 ExampleDS | (Female,FemaleId,{FemaleId,FName})
 ExampleDS | (Male,MaleId,{MaleId,MName})
```

Figure 3.9 shows the mappings between the two local relations and the "Female" domain relation. Each of these mappings contain two correspondences. From the "Employee" local relation there is a correspondence between the "EmployeeId" local attribute and the "FemaleId" domain attribute; and, a correspondence between the "EmployeeName" local attribute and the "FName" domain attribute. With

this mapping, we add the condition that for each of these mappings the "Gender" local attribute has to be equal to 'f'. For the "Student" local relation, there is a correspondence between the "StudentId" local attribute and the "FemaleId" domain attribute; and, a correspondence between the "StudentName" local attribute and the "FName" domain attribute. The conditional mapping visual syntax is used to denote the condition that for the mapping, the "Sex" local attribute has to be equal to 'female'.

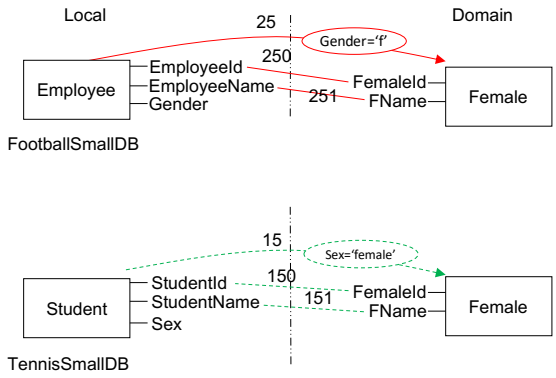


Figure 3.9: Domain relation-local relation mappings are shown between the "Female" domain relation and the "Employee" and "Student" local relations respectively. The upper mapping contains two correspondences between "EmployeeId" and "FemaleId" and "EmployeeName" and "FName". The mapping has the condition that the "Gender" local attribute value must be equal to 'f'. The lower mapping contains two correspondences between "StudentId" and "FemaleId" and "StudentName" and "FName". This mapping has the condition that the "Sex" local attribute value must be equal to 'female'.

Figure 3.10 shows the mappings between the two local relations and the "Male" domain relation. Each of these mappings contain two correspondences. From the "Employee" local relation, there is a correspondence between the "EmployeeId" local attribute and the "MaleId" domain attribute; and, a correspondence between the "EmployeeName" local attribute and the "MName" domain attribute. With these correspondences, the visual conditional correspondence is used to add the

condition that for the mapping the “Gender” local attribute has to be equal to ‘m’. For the “Student” local relation, there is a correspondence between the “StudentId” local attribute and the “MaleId” domain attribute; and, a correspondence between the “StudentName” local attribute and the “MName” domain attribute. The conditional correspondence visual syntax is used to denote the condition that for this mapping, the “Sex” local attribute has to be equal to ‘male’.

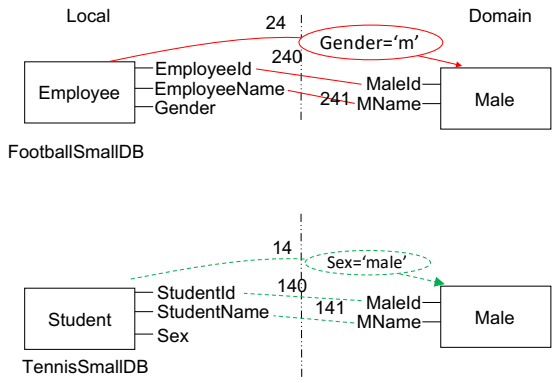


Figure 3.10: Domain relation-local relation mappings are shown for the “Male” domain relation to the “Employee” and “Student” local relations respectively. The upper mapping contains two correspondences between “EmployeeId” and “MaleId” and “EmployeeName” and “MName”. The mapping has the condition that the “Gender” local attribute value must be equal to ‘m’. The lower mapping contains two correspondences between “StudentId” and “MaleId” and “StudentName” and “MName”. This mapping has the condition that the “Sex” local attribute value must be equal to ‘male’.

The correspondences for both domain relations are then represented as shown below in Result Set 3.39. In the first mapping, for the football database (“3”), there are two correspondences between the “Employee” local relation and the “Male” domain relation: correspondence “240” between the “EmployeeId” local attribute and the “MaleId” domain attribute with the condition “gender=‘m’”; and, correspondence “241” between the “EmployeeName” local attribute and the “MName” domain attribute with the condition “gender=‘m’”. In the second mapping, for the

football database ("4"), there are two correspondences between the "Employee" local relation and the "Female" domain relation: correspondence "250" between the "EmployeeId" local attribute and the "FemaleId" domain attribute with the condition "gender='f'"; and, correspondence "251" between the "EmployeeName" local attribute and the "FName" domain attribute with the condition "gender='f'". In the third mapping, for the tennis database ("1"), there are two correspondences between the "Student" local relation and the "Male" domain relation: correspondence "140" between the "StudentId" local attribute and the "MaleId" domain attribute with the condition "sex='male'"; and, correspondence "141" between the "StudentName" local attribute and the "MName" domain attribute with the condition "sex='male'". In the fourth mapping, for the tennis database ("2"), there are two correspondences between the "Student" local relation and the "Female" domain relation: correspondence "250" between the "StudentId" local attribute and the "FemaleId" domain attribute with the condition "gender='f'"; and, correspondence "251" between the "StudentName" local attribute and the "FName" domain attribute with the condition "sex='female'".

Result Set 3.39

```
# select * from ds_ldb_m;
 id |      ldbid      |   dsid    |                              dr_lr_ms
----+-----------------+-----------+-----------------------------------------------------------------
 3  | FootballSmallDB | ExampleDS | (24,Employee,Male,gender='m',{(240,EmployeeId,MaleId),
                                                                  (241,EmployeeName,MName')})
 4  | FootballSmallDB | ExampleDS | (25,Employee,Female,gender='f',{(251,EmployeeName,FName),
                                                                    (250,EmployeeId,FemaleId)})
 1  | TennisSmallDB   | ExampleDS | (14,Student,Male,sex='male',{(141,StudentName,MName),
                                                                  (140,StudentId,MaleId)})
 2  | TennisSmallDB   | ExampleDS | (15,Student,Female,sex='female',{(150,StudentId,FemaleId),
                                                                     (151,StudentName,FName)})
```

Functionally, this case is very similar to the simple mapping example above. The only part of the *apply* operator that will be used for the first time here is in the **proj_type_nest** function (Function 3.1.4 in Equation 3.1); the select operator after the **table_scan** operation will filter the results of the **table_scan** by the predicate p within the mapping. For the "Female" domain relation, the results of the **proj_type_nest** function for the "FName" domain attribute for the tennis database are shown below in Result Set 3.40 and in Result Set 3.41 for the

football database. The results in each have been filtered by "sex='female'" and "gender='f'" respectively.

Result Set 3.40

```
# select * from projtypenest_id('FName','TennisSmallDB','15');
             id              |     fname(value,meta(mid,cid,type))
-----------------------------+-----------------------------------------------
 TennisSmallDB.Student.15.800 | {(Callie Reese,(15,151,StudentName))}
 TennisSmallDB.Student.15.802 | {(Aiko Sweet,(15,151,StudentName))}
```

Result Set 3.41

```
# select * from projtypenest_id('FName','FootballSmallDB','25');
               id                |    fname(value,meta(mid,cid,type))
---------------------------------+-------------------------------------------------
 FootballSmallDB.Employee.25.4187 | {(Mary Stone,(25,251,EmployeeName))}
 FootballSmallDB.Employee.25.7040 | {(Amelia Little,(25,251,EmployeeName))}
 FootballSmallDB.Employee.25.9760 | {(Raja Ryan,(25,251,EmployeeName))}
```

The "FemaleId" domain attribute is produced in a similar fashion and there are no changes in how the **mapped** or *apply* operators behave. The result of the *apply* operator on the "Female" domain relation is shown below in Result Set 3.42.

Result Set 3.42

```
# select * from alpha_id('Female','8');
             id              |              fname                    |         femaleid
-----------------------------+---------------------------------------+--------------------------
TennisSmallDB.Student.15.800|{(Callie Reese,(15,151,StudentName))}  |{(800,(15,150,StudentId))}
TennisSmallDB.Student.15.802|{(Aiko Sweet,(15,151,StudentName))}    |{(802,(15,150,StudentId))}
Foot...DB.Employee.25.4187  |{(Mary Stone,(25,251,EmployeeName))}   |{(4187,(25,250,EmployeeId))}
Foot...DB.Employee.25.7040  |{(Amelia Little,(25,251,EmployeeName))}|{(7040,(25,250,EmployeeId))}
Foot...DB.Employee.25.9760  |{(Raja Ryan,(25,251,EmployeeName))}    |{(9760,(25,250,EmployeeId))}
```

In the same fashion, for the "Male" domain relation, the **proj_type_nest** function (Function 3.1.4 in Equation 3.1) will filter data from the local relations by "sex='male'" and "gender='m'" shown below in Result Sets 3.43 and 3.44.

Result Set 3.43

```
# select * from projtypenest_id('MName','TennisSmallDB','14');
             id              |    mname(value,meta(mid,cid,type))
-----------------------------+-----------------------------------------------
 TennisSmallDB.Student.14.801 | {(Kibo Nolan,(14,141,StudentName))}
 TennisSmallDB.Student.14.803 | {(Elton Duncan,(14,141,StudentName))}
 TennisSmallDB.Student.14.804 | {(Macaulay Hess,(14,141,StudentName))}
```

Result Set 3.44

```
# select * from projtypenest_id('MName','FootballSmallDB','24');
               id                |      mname(value,meta(mid,cid,type))
---------------------------------+-------------------------------------------------
 FootballSmallDB.Employee.24.1578 | {(Dylan Miles,(24,241,EmployeeName))}
 FootballSmallDB.Employee.24.5271 | {(Hasad Wagner,(24,241,EmployeeName))}
```

The result of the *apply* operator on the "Male" domain relation is shown below in Result Set 3.45.

Result Set 3.45

```
# select * from alpha_id('Male','7');
          id                |                mname                  |            maleid
----------------------------+---------------------------------------+--------------------------
TennisSmallDB.Student.14.801|{(Kibo Nolan,(14,141,StudentName))}    |{(801,(14,140,StudentId))}
TennisSmallDB.Student.14.803|{(Elton Duncan,(14,141,StudentName))}  |{(803,(14,140,StudentId))}
TennisSmallDB.Student.14.804|{(Macaulay Hess,(14,141,StudentName))}|{(804,(14,140,StudentId))}
Foot...DB.Employee.24.1578  |{(Dylan Miles,(24,241,EmployeeName))}  |{(1578,(24,240,EmployeeId))}
Foot...DB.Employee.24.5271  |{(Hasad Wagner,(24,241,EmployeeName))}|{(5271,(24,240,EmployeeId))}
```

3.5.5 Combinations

The examples above represent typical mapping scenarios within our system. The comprehensive set of mapping types we allow in our system (as described in our previous chapter) is any combination of the above mapping scenarios. A single local-database-to-domain-structure mapping may contain simple mappings, one local attribute mapped to multiple domain attributes, multiple local attributes mapped to one domain attribute, and conditional correspondences. A domain-structure-application may contain multiple different combinations of these mappings as well. The *apply* operator handles all these cases by considering each one separately.

The first union operation in the *apply* operator causes each separate local-relation-to-domain-relation mapping to be considered separately. Then we process each domain attribute in a separate **proj_type_nest** function and they are then joined back together, allowing the same local attribute to be mapped to multiple domain attributes. This process of separating and joining does add flexibility to our query interface, but it comes at the cost of doing the joins; we will discuss later how simpler mappings can avoid this step. Then, the union operation within each **proj_type_nest** function processes each correspondence separately, letting multiple local attributes be mapped to the same domain attribute.

3.6 CANONICAL APPLY

In order to perform a similar *tablescan*-like operation at the canonical level, we define *canonical apply* (θ), which creates queries against mapped domain structures that will return results from mapped local databases. For example, in the bottom section of Figure 3.5, a *canonical apply* is used against the "Subject" canonical structure to produce the results shown under the box labeled "Canonical (Underlying)". Like the *apply* operation, the *canonical apply* operation returns data

in the format of the canonical relation using the (underlying) nested relational model. The resultant nested relation can then be used with standard nested-relational-algebra operations.

Given a canonical relation named *cr* and a canonical structure application id *csaid*, the *canonical apply* operator is defined in Equation 3.2.

We will explain the steps of the operator in the following example, based on the mapping shown in the top portion of Figure 3.11. In this mapping, there is a canonical relational named "Subject" that contains three canonical attributes ("SubjectId", "SubjectName", "SubjectDetail"). The "Subject" canonical structure is mapped to the "Person" domain relation used previously in this chapter. The mapping contains correspondences between the "SubjectId" canonical attribute and the "PersonId" domain attribute as well as the "SubjectName" canonical attribute and the "GivenName" domain attribute. The "SubjectDetail" canonical attribute is unmapped.

Before we get into the details of the *canonical apply* operator, we describe the example using the implementation.

The canonical structure is described below in Result Set 3.46. The "ExampleCS" canonical structure has a single canonical relation "Subject" with three canonical attributes ("SubjectId", "SubjectName", and "SubjectDetail").

Result Set 3.46

```
# select * from cs;
      id      |                              crs
--------------+-------------------------------------------------------------
 ExampleCS    | (Subject,SubjectId,{SubjectId,SubjectName,SubjectDetail})
```

The domain structure is described below in Result Set 3.47. The "ExampleDS" domain structure has a single domain relation "Person" with two domain attributes ("PersonId" and "GivenName").

Result Set 3.47

```
# select * from ds;
    id     |              drs
-----------+-----------------------------------------
 ExampleDS | (Person,PersonId,{PersonId,GivenName})
```

The mapping shown in the top portion of Figure 3.11 is shown below in Result

$$\theta(cr, csaid) = \bigcup_{\substack{(dsaid, cr\_dr\_m) \in \\ \textbf{csa\_mappings}(cr, csaid)}} \Big(\textbf{cs\_mapped}(dsaid, cr\_dr\_m) \times \textbf{cs\_not\_mapped}(cr, cr\_dr\_m, csaid) \Big) \tag{1}$$

$$\textbf{csa\_mappings}(cr, csaid) = \pi_{\substack{csa.dsaid, \\ cs\_ds\_m.cr\_dr\_ms}} \left(\sigma_{\substack{csa.id = csaid \\ \wedge cs.crs.name = cr}} \Big(cs\_ds\_m \underset{cs\_ds\_m.id = csa.cs\_ds\_mid}{\bowtie} (\mu_{cs\_ds\_ms}(csa)) \Big) \right) \tag{2}$$

$$\textbf{cs\_mapped}(dsaid, cr\_dr\_m) = \pi_{id, \left\{ [da] \rightarrow [ca] \mid (da, ca) \in \pi_{\substack{cr\_dr\_m.corrs.da, \\ cr\_dr\_m.corrs.ca}} (cr\_dr\_m) \right\}} (\alpha(cr\_dr\_m.dr, dsaid)) \tag{3}$$

$$\textbf{cs\_not\_mapped}(cr, cr\_dr\_m, csaid) = \bigtimes_{\substack{ca \in \textbf{cattrs}(cr, csaid) \wedge \\ \textbf{is\_empty}(\sigma_{cr\_dr\_m.corrs.ca = [ca]}(cr\_dr\_m))}} \underset{\rightarrow [ca](value, meta(mid, cid, type))}{(NULL, (NULL, NULL, NULL))} \tag{4}$$

$$\textbf{cattrs}(cr, csaid) = \underset{cs.crs.attrs}{\pi} (\sigma_{\substack{csa.id = [csaid] \\ \wedge cs.crs.name = [cr]}} (cs \underset{cs.id = csa.csid}{\bowtie} csa)) \tag{5}$$

Equation 3.2: Canonical Apply

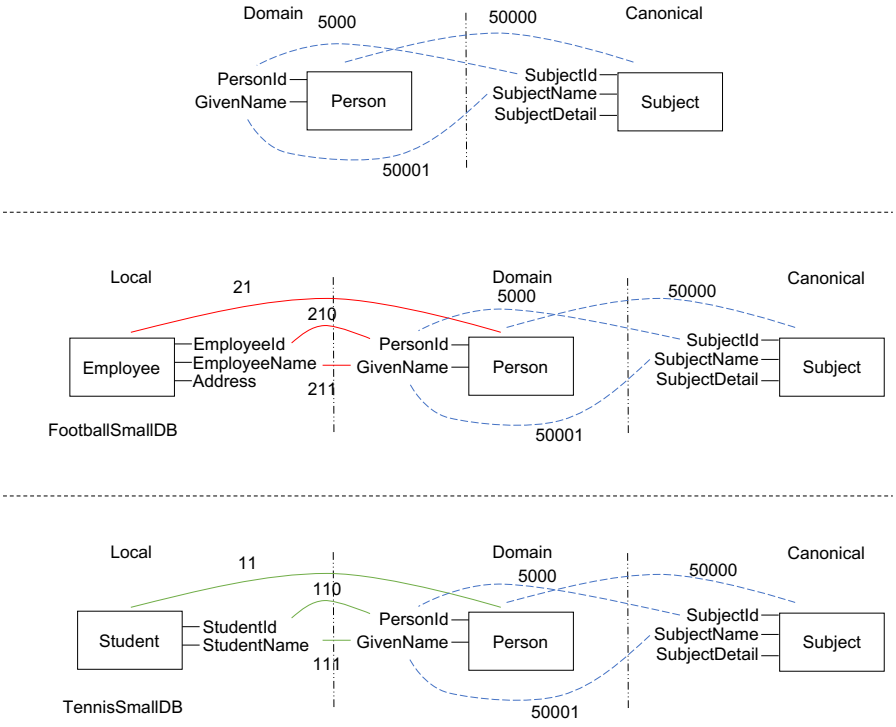


Figure 3.11: A canonical relation-domain relation mapping is shown (top) with added domain relation-local relation mappings shown (middle and bottom).

Set 3.48. In the domain-structure-to-canonical-structure mapping with id "500", there exists one mapping, "5000", between the "Person" domain relation and the "Subject" canonical relation that contains two correspondences ("50000", between the "PersonId" domain attribute and the "SubjectId" canonical attribute, and "50001", between the "GivenName" domain attribute and the "SubjectName" canonical attribute).

Result Set 3.48

```
# select * from cs_ds_m;
 id  |   dsid    |     csid     |                          cr_dr_ms
-----+-----------+--------------+-------------------------------------------------------------
 500 | ExampleDS | ExampleCS    | (5000,Person,Subject,{(50000,PersonId,SubjectId),
                                (50001,GivenName,SubjectName)})
```

The local databases are described below in Result Set 3.49. The "FootballSmallDB" database has a single relation, "Employee", with two attributes ("EmployeeId" and "EmployeeName"). The "TennisSmallDB" has a single relation

"Student" with two attributes ("StudentId" and "StudentName").

Result Set 3.49

```
#select * from ldb;
       id        |                      lrs
-----------------+-------------------------------------------------
 FootballSmallDB | (Employee,EmployeeId,{EmployeeId,EmployeeName})
 TennisSmallDB   | (Student,StudentId,{StudentId,StudentName})
```

The local-database-to-domain-structure mappings in the middle and bottom portions of Figure 3.11 are shown below in Result Set 3.50. The red mapping "21" shown in the middle portion of the Figure is between the "Employee" local relation and the "Person" domain relation with correspondences "210" (between the "EmployeeId" local attribute and the "PersonId" domain attribute) and "211" (between the "EmployeeName" local attribute and "GivenName" domain attribute). The green mapping "11" shown in the bottom of Figure 3.11 is between the "Student" local relation and the "Person" domain relation, with correspondences "110" (between the "StudentId" local attribute and the "PersonId" domain attribute) and "111" (between the "StudentName" local attribute and "GivenName" domain attribute).

Result Set 3.50

```
# select * from ds_ldb_m;
 id |      ldbid      |   dsid    |                          dr_lr_ms
----+-----------------+-----------+-------------------------------------------------------------
 1  | TennisSmallDB   | ExampleDS | (11,Student,Person,TRUE,{(110,StudentId,PersonId),
                                                          (111,StudentName,GivenName)})
 2  | FootballSmallDB | ExampleDS | (21,Employee,Person,TRUE,{(210,EmployeeId,PersonId),
                                                          (211,EmployeeName,GivenName)})
```

In this example we have a single domain-structure-application "3" that contains mappings "1" and "2", shown below in Result Set 3.51.

Result Set 3.51

```
# select * from dsa;
 id |   dsid    | ds_ldb_mid
----+-----------+------------
 3  | ExampleDS | {1,2}
```

There is a single canonical-structure-application that contains the domain structure application "3" from above and the domain-structure-to-canonical-structure mapping "500", shown below in Result Set 3.52. Note that the "cs_ds_ms" attribute is nested, since there may be multiple canonical structure to domain structure mappings used in a single canonical-structure application.

Result Set 3.52

```
# select * from csa;
 id    |     csid     | dsaid | cs_ds_ms
-------+--------------+-------+----------
 csa3  | ExampleCS    | 3     | {500}
```

Now we describe the details of the *canonical apply* operator. Given the canonical relation "Subject" and the canonical structure application id "csa3", we begin with Function 3.2.1, where θ(Subject, csa3) will return the cross product of the **cs_mapped** and **cs_not_mapped** functions on all the domain-structure application and domain-relation-to-canonical-relation mapping tuples returned from the **csa_mappings** function (3.22) and union their results.

Given the canonical relation "Subject" and the canonical-application id "csa3", the **csa_mappings** function (Function 3.2.2) unnests the "cs_ds_ms" attribute from the "csa" relation (shown in Result Set 3.52) and joins that with the "cs_ds_m" relation (shown in Result Set 3.48) using the mapping id. That result is then filtered by the input canonical relation and canonical-structure-application id and the domain-structure-application id from the "csa" relation. The domain-relation-to-canonical-relation mappings ("cr_dr_ms") from the "cs_ds_m" relation are projected into the result, shown below in Result Set 3.53.

Result Set 3.53

```
# select * from mapping_applications('Subject','csa3');
 dsaid |                                cr_dr_ms
-------+-----------------------------------------------------------
 3     | (5000,Person,Subject,{(50000,PersonId,SubjectId),
                               (50001,GivenName,SubjectName)})
```

Using the domain-structure-application id and domain-relation-to-canonical-relation mappings returned from **csa_mappings**, the *canonical apply* operation returns the local data for all mapped canonical attributes using the **cs_mapped** function (Function 3.2.3). This function begins by performing the *apply* operation on the domain relation specified in the input domain-relation-to-canonical-relation mapping ("Person") using the input domain-structure-application id ("3"). Assuming the local data for the "Student" and "Employee" relations shown below

in Result Sets 3.54 and 3.55, the result of the *apply* operation is shown in Result Set 3.56.

Result Set 3.54

```
# select * from student;
 studentid |  studentname
-----------+---------------
 800       | Callie Reese
 801       | Kibo Nolan
```

Result Set 3.55

```
# select * from employee;
 employeeid | employeename
------------+---------------
 9760       | Raja Ryan
 4187       | Mary Stone
```

Result Set 3.56

```
# select * from alpha_id('Person','3');
                id                |          personid          |           givenname
----------------------------------+----------------------------+----------------------------------
 TennisSmallDB.Student.11.800     |{(800,(11,110,StudentId))}  |{(Call...,(11,111,StudentName))}
 TennisSmallDB.Student.11.801     |{(801,(11,110,StudentId))}  |{(Kibo...,(11,111,StudentName))}
 FootballSmallDB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...,(21,211,EmployeeName))}
 FootballSmallDB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...,(21,211,EmployeeName))}
```

The **cs_mapped** function (Functon 3.2.3) then renames all the domain attributes to canonical attributes using the tuples of domain-attribute-canonical-attribute pairs projected from the input domain-relation-to-canonical-relation mapping. The function projects the id and canonical attributes. The results are shown below in Result Set 3.57.

Result Set 3.57

```
# select * from cs_mapped('3','Subject','5000');
                id                |          subjectid         |          subjectname
----------------------------------+----------------------------+----------------------------------
TennisSmallDB.Student.11.800     |{(800,(11,110,StudentId))}  |{(Call...,(11,111,StudentName))}
TennisSmallDB.Student.11.801     |{(801,(11,110,StudentId))}  |{(Kibo...,(11,111,StudentName))}
FootballSmallDB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...,(21,211,EmployeeName))}
FootballSmallDB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...,(21,211,EmployeeName))}
```

Using the domain-structure-application id and domain-relation-to-canonical-relation mappings returned from **csa_mappings**, the *canonical apply* operation then adds any canonical attributes that were not contained in the mapping, using the **cs_not_mapped** function (3.2.4). The **cs_not_mapped** function finds unmapped canonical attributes by finding all the attributes in the canonical relation using the **cattrs** function (3.2.5) where there are no correspondences in the input mapping (using the **is_empty** function while selecting the canonical attributes from the mapping). In this case, we have one canonical attribute that was not mapped ("SubjectDetail"), so the function will create a nested attribute named

"SubjectDetail" with "NULL" values for the "value", "mid", "cid", and "type" attributes as shown in Result Set 3.58.

Result Set 3.58

```
# select * from cs_not_mapped('3','5000','csa3');
 subjectdetail(value,meta(mid,cid,type))
-----------------------------------------
 {(NULL,(NULL,NULL,NULL))}
```

The last step in the *canonical apply* is to take the cross product of the **cs_mapped** and **cs_not_mapped** functions, the result is shown below in Result Set 3.59.

Result Set 3.59

```
# select * from theta('Subject','csa3');
       id       |         subjectid         |          subjectname             |  subjectdetail
----------------+---------------------------+----------------------------------+-----------------
 Tennis...11.800| {(800,(11,110,StudentId))}| {(Call...,(11,111,StudentName))}| {(NULL,(NULL...}
 Tennis...11.801| {(801,(11,110,StudentId))}| {(Kibo...,(11,111,StudentName))}| {(NULL,(NULL...}
 Foot...21.4187 | {(4187,(21,210,Emp...Id))}| {(Mary...,(21,211,Emp...Name))} | {(NULL,(NULL...}
 Foot...21.9760 | {(9760,(21,210,Emp...Id))}| {(Raja...,(21,211,Emp...Name))} | {(NULL,(NULL...}
```

3.7 APPARENT MODEL AND TYPE OPERATIONS

We have shown how the *apply* and *canonical apply* operators work. Both operators return a nested relational result in the underlying model, but recall from Figure 3.5 that to meet our goal of simplicity we do not expect our end users to interact with or understand the nested relational model; we instead provide the apparent model. To do that we define the *apparent model* operator (κ) shown in Equation 3.3. The *apparent model* operator will work on the result of either an *apply* or *canonical apply* operator, since both have the same format. We focus on the definition of the *apparent model* operator using the *canonical apply*, as that is the most likely use case.

$$\kappa(cr, csaid) = \pi_{id,[ca_1],\ldots,[ca_n]}(\underset{value\rightarrow[ca_1]}{\rho}(\mu_{[ca_1]}(\ldots(\underset{value\rightarrow[ca_n]}{\rho}(\mu_{[ca_n]}(\theta(cr, csaid)))))))$$

$$\forall ca_i \in \mathbf{cattrs}(cr, csaid) \tag{3.3}$$

Recalling that **cattrs** is defined in Function 3.2.5 above as

$$\mathbf{cattrs}(cr, csaid) = \underset{cs.crs.attrs}{\pi}(\sigma_{\substack{csa.id=csaid \\ \wedge cs.crs.name=cr}}(cs \underset{cs.id=csa.csid}{\bowtie} csa))$$

The *apparent model* operator functions like a *tablescan* operator at the canonical level in a query, returning a relational result that can then be combined with any standard relational operators. The *apparent model* operator takes a canonical relation name and a canonical-structure-application id, then produces the result of the *canonical apply* for those inputs, unnests each canonical attribute and renames the "value" attribute to the canonical attribute name and projects the id and the renamed attributes. For example, the result of the *apparent model* operator on the canonical relation "Subject" with the canonical structure application id "csa3" will first run the *canonical apply*, producing the results above in Result Set 3.59. The renamed and projected result would then be as follows in Result Set 3.60.

Result Set 3.60

```
# make_apparent('Subject','csa3');
             id               | subjectid |   subjectname | subjectdetail
------------------------------+-----------+---------------+--------------
 TennisDB.Student.30.800      | 800       | Callie Reese  | NULL
 TennisDB.Student.30.801      | 801       | Kibo Nolan    | NULL
 FootballDB.Employee.70.4187 | 4187      | Mary Stone    | NULL
 FootballDB.Employee.70.9760 | 9760      | Raja Ryan     | NULL
```

A consequence of the unnesting in the *apparent model* operator is that the id attribute may no longer be a key for the resulting relation. Such is the case when multiple local attributes are mapped to the same domain attribute, resulting in nested tuples in the domain attribute. For example, suppose the "Person" domain structure with the "Organizational Unit" domain attribute is mapped to the "Subject" canonical structure, as shown in Figure 3.12. The result from the *apply* operator was described above and shown in Result Set 3.34. The result of the *canonical apply* operator would be as follows in Result Set 3.61.

Result Set 3.61

```
# select * from theta('Person','csa3');
             id           |            subjectid       |subjectname|     subjectdetail
--------------------------+----------------------------+-----------+-----------------------------
Tennis...DB.Student.11.800|{(800,(11,110,StudentId))}  |{(Call...  |{(physics,(11,112,Depar...))}
Tennis...DB.Student.11.801|{(801,(11,110,StudentId))}  |{(Kibo...  |{(math,(11,112,Depar...))}
Foot...DB.Employee.21.4187|{(4187,(21,210,EmployeeId))}|{(Mary...  |{(america,(21,213,Region)),
                                                                     (league 2,(21,214,Division))}
Foot...DB.Employee.21.9760|{(9760,(21,210,EmployeeId))}|{(Raja...  |{(europe,(21,213,Region)),
                                                                     (league 1,(21,214,Division))}
```

The result of the *apparent model* operator on this instance of the "Subject" canonical structure is then as follows in result Set 3.62.

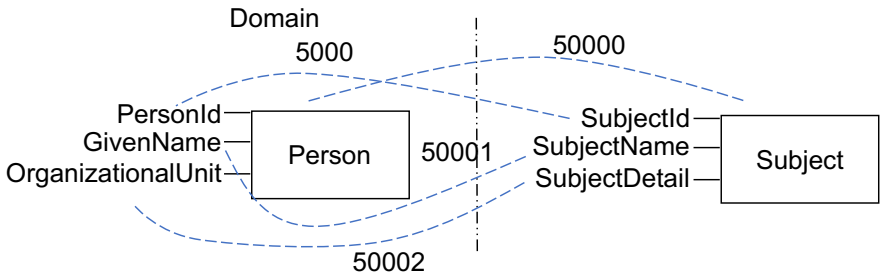


Figure 3.12: A domain-relation-to-canonical-relation mapping is shown from the "Person" domain structure (containing the "Organizational Unit" domain attribute) to the "Subject" canonical structure.

Result Set 3.62

```
# make_apparent('Subject','csa3');
             id              | subjectid |   subjectname | subjectdetail
-----------------------------+-----------+---------------+--------------
 TennisDB.Student.30.800     | 800       | Callie Reese  | physics
 TennisDB.Student.30.801     | 801       | Kibo Nolan    | math
 FootballDB.Employee.70.4187 | 4187      | Mary Stone    | america
 FootballDB.Employee.70.4187 | 4187      | Mary Stone    | league 2
 FootballDB.Employee.70.9760 | 9760      | Raja Ryan     | europe
 FootballDB.Employee.70.9760 | 9760      | Raja Ryan     | league 1
```

In order to radiate local information to the canonical level in the apparent model we then also define the *type* operator, τ, to extract local type information. The *type* operator (Equation 3.4) takes a canonical-relation name "cr", a canonical-attribute name "ca", and a canonical-structure-application id "csaid" and returns a relation containing an id, the value for that id, and the local type information. This result can then be used alone or combined with the *canonical apply* operator (like the query in Figure 3.5 where the natural join is used to combine the type operator with the results of a *canonical apply* operator) to show local data and type information.

$$\tau(cr, ca, csaid) = \pi_{id,value,type}(\mu_{meta}(\mu_{[ca]}(\theta(cr, csaid)))) \tag{3.4}$$

The operator takes the given canonical relation and canonical-structure-application id, runs the *canonical apply* for those inputs, unnests the given canonical attribute and its nested "meta" attribute, and then projects out the "id", "value", and "type" attributes. Continuing with the example above, the *type*

operator on the "Subject" canonical relation, for the "SubjectName" canonical attribute, with canonical structure application id "csa3" produces the result shown in Result Set 3.63.

Result Set 3.63

```
# select * from tau('Subject','SubjectName','csa3');
                id                 |     value     |     type
-----------------------------------+---------------+--------------
 TennisSmallDB.Student.11.800      | Callie Reese  | StudentName
 TennisSmallDB.Student.11.801      | Kibo Nolan    | StudentName
 FootballSmallDB.Employee.21.4187 | Mary Stone    | EmployeeName
 FootballSmallDB.Employee.21.9760 | Raja Ryan     | EmployeeName
```

We also want to provide access to the local relation name (in addition to the local attribute names). We overload the *type* operator and define a version that only takes two parameters (as opposed to the three above). This version takes the canonical relation and canonical-structure-application id as parameters and returns the id and local relation name. This version of the operator allows the local type information for the relation (i.e., the relation name) to be radiated to the canonical level.

$$\tau(cr, csaid) = \pi_{id, split(id,'.')[1] \rightarrow type}(\theta(cr, csaid)) \tag{3.5}$$

This version of the type operator projects the id from the canonical apply and then extracts the local relation type from the id using the split function (which returns an array of strings by splitting an input string on a given delimiter—in this case the period) and then renames the second element of the split array to "type". Using the example above, the local type information for the canonical relation "SubjectName" would be:

Result Set 3.64

```
# select * from tau('Subject','csa3');
               id                 |  type
----------------------------------+----------
TennisSmallDB.Student.11.800      | Student
TennisSmallDB.Student.11.801      | Student
FootballSmallDB.Employee.21.4187 | Employee
FootballSmallDB.Employee.21.9760 | Employee
```

3.8 OPTIMIZATIONS

While our system introduces overhead by processing all mappings individually and adding information about mappings, correspondences, and types, the naive implementation based on the formalism can be optimized for faster query processing. We introduce modified versions of the *apply* and *canonical apply* operators that allow common relational equivalences to be performed by an optimizer. For example a query that projects only the "SubjectName" from the "Subject" canonical relation in Figure 3.12, i.e., $\pi_{SubjectName}(\theta(Subject))$, need not process the "SubjectDetail" canonical attribute, which in turn means we need not process the "OrganizationalUnit" domain attribute from the "Person" domain relation. Similarly, if we have a selection query, such as $\sigma_{SubjectName=\text{`Mary Stone'}}(\theta(Subject))$, we want to push that selection predicate down to the individual local databases so that we only have to bring the relevant tuples to the domain and canonical levels. We present our new operators and equivalences to be used with them. We expect that the use of the optimized operators will be done by a query optimizer and not query writers, as we still expect query writers to either be working in an apparent model or be using the standard form of the operator. We also present an *optimized apply* operator that eliminates many of the joins in the standard *optimized apply* operator in certain mapping cases, which will be described below.

3.8.1 Optimized Apply

The optimized version of *apply* shown in Equation 3.6 allows projection and selection operators to be pushed into *apply*. Changes to the original operator are shown in red in the formalism. The modified operator takes additional parameters *pred* and $pattrs$. The *pred* parameter[5] is used to pass the predicate of a selection into

[5]Note that this is a predicate from a selection query being pushed into the *apply* operator and different from the predicate that may be supplied with a mapping.

the operator; we require that only mapped domain attributes are referenced in the predicate. The *pattrs* parameter passes the projection list into the operator; we require that the attribute list contains only domain attributes for the given domain relation, *dr*, but may contain mapped or unmapped attributes.

In the first line of the operator (Equation 3.6), the new parameters are passed along to the $\mathbf{mp}_{opt}$ function (which takes both parameters) and the $\mathbf{nmp}_{opt}$ function (which only takes the *pattrs* parameter since we assume the predicate does not reference unmapped attributes)[6].

The **dsa_mappings** function (Function 3.6.2 in Equation 3.6) remains unchanged from the definition in Equation 3.1.

The $\mathbf{mp}_{opt}$ function (Function 3.6.3) passes the predicate *pred* into each $\mathbf{ptn}_{opt}$ function that is run. Since the $\mathbf{ptn}_{opt}$ function is run for each domain attribute in the domain relation, we use the predicate list *pattrs* to run only the $\mathbf{ptn}_{opt}$ functions for domain attributes appearing in the attribute list, avoiding unnecessary work. Note that if the developer wants all the attributes then there will be no project and the predicate list will be empty, i.e., $pattrs = NULL$. In that case, $\mathbf{ptn}_{opt}$ is run for all mapped domain attributes.

The $\mathbf{ptn}_{opt}$ function (Function 3.6.4) pushes the predicate *pred* into the selection operator directly after the table scan. The **replace_pred** function is used to rename all domain attributes in the predicate to their respective local attribute names. Since we require that all attributes referenced in the predicate have been mapped, we can push this predicate into all $\mathbf{ptn}_{opt}$ functions, as they all reference the same local relation for a given mapping. This prevents us from having to determine which term in the predicate needs to be sent to each $\mathbf{ptn}_{opt}$ function.

The predicate list *pattrs* is used in the $\mathbf{nmp}_{opt}$ function (Function 3.6.7) to limit

[6]For the sake of brevity in Equation 3.6 the function names have been abbreviated from their names in the *apply* operator (Equation 3.1); **mapped** to **mp**, **not_mapped** to **nmp**, **proj_type_nest** to **ptn**, and **dr_lr_m** to **dlm**.

the number of attributes created by only adding attributes to the cross product if the domain attribute is in the attribute list $pattrs$.

Note that if the *pred* parameter is *true* and *pattr* parameter is empty (= $NULL$), the operator functions the same as the original *apply* operator.

Using the *optimized apply* operator we propose the following equivalences to facilitate query execution and optimization. First we show how a relational projection operator can be pushed into the *optimized apply*.

Theorem 3.1. $\pi_{pattrs}(\alpha(dr, dsaid)) \equiv \alpha_{opt}(dr, dsaid, \boldsymbol{true}, pattrs)$

In the following proof (and those that follow in this section) we abbreviate parts of the functions that remain unchanged by the steps of the proof. Justifications for each step of the proof begin on the line below each numbered proof statement.

Proof.

$$\pi_{pattrs}(\alpha(dr, dsaid)) \equiv \pi_{pattrs}(\bigcup(\mathbf{mp} \bigtimes \mathbf{nmp})) \tag{1}$$

by the definition of the *apply* operator, Equation 3.1.

$$\equiv \bigcup(\pi_{pattrs}(\mathbf{mp}) \bigtimes \pi_{pattrs}(\mathbf{nmp})) \tag{2}$$

by the equivalences $\pi_{\omega}(E_0 \cup E_1) \equiv \pi_{\omega}(E_0) \cup \pi_{\omega}(E_1)$

and $\pi_{\omega}(E_0 \times E_1) \equiv \pi_{\omega}(E_0) \times \pi_{\omega}(E_1)$ [1, 26].

$$\equiv \bigcup(\pi_{pattrs}(\mathbf{ptn}(da_1) \bowtie \ldots \bowtie \mathbf{ptn}(da_n)) \bigtimes \pi_{pattrs}(\mathbf{nmp})) \tag{3}$$

by the definition of the **mapped** function (Function 3.1.3).

$$\equiv \bigcup((\pi_{pattrs}\mathbf{ptn}(da_1) \bowtie \ldots \bowtie \pi_{pattrs}\mathbf{ptn}(da_n)) \bigtimes \pi_{pattrs}(\mathbf{nmp})) \tag{4}$$

by the equivalence $\pi_{\omega}(E_0 \bowtie E_1) \equiv \pi_{\omega}(E_0) \bowtie \pi_{\omega}(E_1)$ [1, 26].

$$\equiv \bigcup((\bowtie \mathbf{ptn}(da_i) | da_i \in pattrs) \bigtimes \pi_{pattrs}(\mathbf{nmp})) \tag{5}$$

$$\alpha_{opt}(dr, dsaid, pred, pattrs) = \bigcup_{\substack{(ldbid,dlm)\in \\ \textbf{dsa\_mappings}(dr,dsaid)}} \Big(\textbf{mp}_{opt}(ldbid, dlm, pred, pattrs) \bigtimes \textbf{nmp}_{opt}(dr, dlm, dsaid, pattrs)\Big) \tag{1}$$

$$\textbf{dsa\_mappings}(dr, dsaid) = \pi_{\substack{ds\_ldb\_m.ldbid, \\ ds\_ldb\_m.dr\_lr\_ms\to dlm}} \left(\sigma_{\substack{ds\_ldb\_m.dr\_lr\_ms.dr=dr \\ \wedge dsa.id=dsaid}} \Big(ds\_ldb\_m \underset{ds\_ldb\_m.id=dsa.ds\_ldb\_mid}{\bowtie} (\mu_{ds\_ldb\_ms}(dsa)) \Big) \right) \tag{2}$$

$$\textbf{mp}_{opt}(ldbid, dlm, pred, pattrs) = \textbf{ptn}_{opt}(da_1, ldbid, dlm, pred) \underset{id}{\bowtie} \ldots \underset{id}{\bowtie} \textbf{ptn}_{opt}(da_n, ldbid, dlm, pred) \tag{3}$$
$$\forall da_i \in \sigma_{da\in pattrs \vee pattrs=NULL}(\pi_{dlm.corrs.da}(dlm))$$

$$\textbf{ptn}_{opt} \quad (da, ldbid, dlm, pred) = \nu_{value,meta:[da]} \left(\bigcup_{corr\in\pi_{dlm.corrs}(\sigma_{dr\_lr\_m.corrs.da=da}(dlm))} \Big(\right. \tag{4}$$
$$\left. \Big(\pi_{\substack{\textbf{gen\_key}(ldbid,dlm)\to id, \\ [corr.la]\to value}} \big(\sigma_{\substack{dlm.p\wedge \\ \textbf{replace\_pred}(pred,dlm)}} (\textbf{table\_scan}(ldbid, dlm.lr))\big)\Big) \bigtimes \underset{\to meta(mid,cid,type)}{(dlm.id, corr.id, corr.la)} \Big) \right)$$

$$\textbf{gen\_key}(ldbid, dlm) = ldbid||.||dlm.lr||.||dlm.id||.||[\pi_{ldb.lrs.key}(\sigma_{\substack{ldb.lrs.name=dlm.lr \\ \wedge ldb.id=ldbid}}(ldb)] \tag{5}$$

$$\textbf{replace\_pred}(pred, dlm) = \textbf{string\_replace}(pred, da, la) \forall (da, la) \in \pi_{\substack{dlm.corrs.da, \\ dlm.corrs.la}}(dlm) \tag{6}$$

$$\textbf{nmp}_{opt}(dr, dlm, dsaid, pattrs) = \bigtimes_{\substack{[da]\in\textbf{dattrs}(dr,dsaid)\wedge \\ \textbf{is\_empty}(\sigma_{dlm.corrs.da=[da]}(dlm))\wedge \\ ([da]\in pattrs \vee pattrs=NULL)}} \underset{\to[da](value,meta(mid,cid,type))}{(NULL, (dlm.id, NULL, NULL))} \tag{7}$$

$$\textbf{dattrs}(dr, dsaid) = \pi_{ds.drs.attrs} \Big(\sigma_{\substack{ds.drs.name=[dr] \\ \wedge dsa.id=[dsaid]}} (ds \underset{ds.id=dsa.dsid}{\bowtie} dsa) \Big) \tag{8}$$

Equation 3.6: Optimized Apply

since the results of the **ptn** function for any da_i not in the project list cannot be in the results of the function by the definition of the project operator.

$$\equiv \bigcup((\bowtie \mathbf{ptn}(da_i)|da_i \in pattrs) \bigtimes \pi_{pattrs}(da_{n+1} \times \ldots \times da_m)) \quad (6)$$

by the definition of the **not_mapped** (**nmp**) function (Function 3.1.6 in Equation 3.1). Here $da_{n+1}, \ldots, da_m$ are the domain attributes that are not in the current mapping and da_n was the last mapped attribute in the **mapped** function.

$$\equiv \bigcup((\bowtie \mathbf{ptn}(da_i)|da_i \in pattrs) \bigtimes (\times da_j|da_j \in pattrs)) \quad (7)$$

since any unmapped da_i not in the project list cannot be in the results of the function by the definition of the project operator. Here $da_j \in \{da_{n+1}, \ldots, da_m\}$ and we use $\times da_j$ to represent the cross-product of all such da.

$$\equiv \alpha_{opt}(dr, dsaid, \mathbf{true}, pattrs) \quad (8)$$

by the definition of the *optimized apply* operator (Equation 3.6).

□

We also provide an equivalence that allows selection predicates to be pushed into the *optimized apply* operator. We assume that the predicate only contains literals or domain attributes from the given dr.

Theorem 3.2. $\sigma_{pred}(\alpha(dr, dsaid)) \equiv \alpha_{opt}(dr, dsaid, pred, NULL)$

Proof.

$$\sigma_{pred} \quad (\alpha(dr, dsaid)) \equiv \sigma_{pred}(\bigcup(\mathbf{mp} \bigtimes \mathbf{nmp})) \quad (1)$$

by the definition of the *apply* operator, Equation 3.1.

$$\equiv \bigcup(\sigma_{pred}(\mathbf{mp}) \bigtimes \mathbf{nmp}) \tag{2}$$

by the equivalences $\sigma_{\omega}(E_0 \cup E_1) \equiv \sigma_{\omega}(E_0) \cup \sigma_{\omega}(E_1)$

and $\sigma_{\omega}(E_0 \times E_1) \equiv \sigma_{\omega}(E_0) \times \sigma_{\omega}(E_1)$ [1, 26] and the requirement that only mapped attributes and literals exist in *pred*.

$$\equiv \bigcup(\sigma_{pred}(\mathbf{ptn}(da_1) \bowtie \ldots \bowtie \mathbf{ptn}(da_n)) \bigtimes \mathbf{nmp}) \tag{3}$$

by the definition of the **mapped** function (Function 3.1.3 in Equation 3.1).

$$\equiv \bigcup((\sigma_{pred}\mathbf{ptn}(da_1) \bowtie \ldots \bowtie \sigma_{pred}\mathbf{ptn}(da_n)) \bigtimes \mathbf{nmp}) \tag{4}$$

by the relational algebra equivalence $\sigma_{\omega}(E_0 \bowtie E_1) \equiv \sigma_{\omega}(E_0) \bowtie \sigma_{\omega}(E_1)$ [1, 26].

$$\equiv \bigcup((\sigma_{pred}(\nu(\bigcup(\pi_{id,la \to da}(\sigma_{corr.p}(lr)))) \bowtie \ldots) \bigtimes \mathbf{nmp}) \tag{5}$$

by the definition of the **proj_type_nest** function (Function 3.1.4).

$$\equiv \bigcup((\sigma_{pred}(\nu(\bigcup(\pi_{id,da}(\rho_{la \to da}(\sigma_{corr.p}(lr))))) \bowtie \ldots) \bigtimes \mathbf{nmp}) \tag{6}$$

by the definition of the project with renaming operator [49].

$$\equiv \bigcup((\nu(\bigcup(\pi_{id,da}(\rho_{la \to da}(\sigma_{\substack{corr.p \wedge \\ \mathbf{replace\_pred}(pred)}}(lr)))) \bowtie \ldots) \bigtimes \mathbf{nmp}) \tag{7}$$

by the definition of the nested selection operator [49], the relational algebra equivalence $\sigma_{\omega}(E_0 \cup E_1) \equiv \sigma_{\omega}(E_0) \cup \sigma_{\omega}(E_1)$, the relational algebra equivalence $\sigma_{\omega}(\pi_{\delta}(E_0)) \equiv \pi_{\delta}(\sigma_{\omega}(E_0))$, and the definition of the rename operator [49], which means that *pred* and **replace_pred**(*pred*) are the same.

$$\equiv \alpha_{opt}(dr, dsaid, pred, NULL) \tag{8}$$

by the definition of the *optimized apply* operator (Equation 3.6).

□

3.8.2 Optimized Canonical Apply

The optimized version of *canonical apply* shown in Equation 3.7 allows projection and selection operators to be pushed into *apply*. Changes to the original operator are shown in red in the formalism. The modified operator takes additional parameters *pred* and $pattrs$. The *pred* parameter is used to pass the predicate of a selection into the operator; we assume here that only mapped canonical attributes are referenced in the predicate. The $pattrs$ parameter passes the projection list into the operator; we assume here that the attribute list only contains canonical attributes for the given canonical relation, cr, but may contain mapped or unmapped attributes.

Both the *pred* and $pattrs$ parameters are then passed into the $\mathbf{cs\_mp}_{opt}$ function[7] while only the $pattrs$ parameter is passed into the $\mathbf{cs\_nmp}_{opt}$ function, since we require that no unmapped canonical attributes are referenced in the predicate.

The **cs_mapped** function (Function 3.2.3) is then modified (Function 3.2.3). A condition is added that is used to build the attribute list of the projection operator such that the canonical attribute must be in the input attribute list $pattrs$ or, if $pattrs$ is empty, then it should operate as before in Equation 3.2. The function is modified to use the *optimized apply* operator. The *pred* and $pattrs$ parameters are transformed to replace all canonical attributes with domain attributes before being passed to the *optimized apply* operator using the **replace_pred** and **replace_pattr** functions.

The **cs_not_mapped** function (Function 3.2.4) is modified in (Function 3.2.4) by adding a condition to the cross product such that only unmapped attributes

[7]As with the *optimized apply* operator we also abbreviate function names here; **cs_mapped** to **cs_mp**, **cs_not_mapped** to **cs_nmp** and **cs_ds_m** to **cdm**.

are created if they exist in *pattrs* or if *pattrs* is empty all unmapped attributes are added.

We provide two relational algebra equivalences that can be used with the *optimized canonical apply* operator. We show how a relational projection operator can be pushed into the *optimized canonical apply*.

Theorem 3.3. $\pi_{pattrs}(\theta(cr, csaid)) \equiv \theta_{opt}(cr, csaid, \textbf{\textit{true}}, pattrs)$

Proof.

$$\pi_{pattrs}(\theta(cr, csaid)) \equiv \pi_{pattrs}(\bigcup(\textbf{cs\_mp} \bigtimes \textbf{cs\_nmp})) \tag{1}$$

by the definition of the *canonical apply* operator, Equation 3.2.

$$\equiv \bigcup(\pi_{pattrs}(\textbf{cs\_mp}) \bigtimes \pi_{pattrs}(\textbf{cs\_nmp})) \tag{2}$$

by the equivalences $\pi_{\omega}(E_0 \cup E_1) \equiv \pi_{\omega}(E_0) \cup \pi_{\omega}(E_1)$

and $\pi_{\omega}(E_0 \times E_1) \equiv \pi_{\omega}(E_0) \times \pi_{\omega}(E_1)$ [1, 26].

$$\equiv \bigcup(\pi_{pattrs}(\pi_{[da_i]\rightarrow[ca_i]}(\alpha(dr, dsaid))) \bigtimes \pi_{pattrs}(\textbf{cs\_nmp})) \tag{3}$$

by the definition of the **cs_mapped** function (Function 3.2.3).

$$\equiv \bigcup(\pi_{pattrs}(\pi_{[ca_i]}(\rho_{[da_i]\rightarrow[ca_i]}(\alpha(dr, dsaid)))) \bigtimes \pi_{pattrs}(\textbf{cs\_nmp})) \tag{4}$$

by the definition of the project with renaming operator [49].

$$\equiv \bigcup(\rho_{[da_i]\rightarrow[ca_i]}(\pi_{pattr'}(\alpha(dr, dsaid))) \bigtimes \pi_{pattrs}(\textbf{cs\_nmp})) \tag{5}$$

by the definition of the rename operator [49] and where

$pattr' = \textbf{replace\_pattr}(pattrs, cdm)$.

$$\equiv \bigcup(\pi_{[ca_i]}(\rho_{[da_i]\rightarrow[ca_i]}(\alpha_{opt}(\ldots, pattr'))) \bigtimes \pi_{pattrs}(\textbf{cs\_nmp})) \tag{6}$$

by Theorem 3.1.

$$\theta_{opt}(cr, csaid, pred, pattrs) = \bigcup_{\substack{(dsaid,cdm)\in \\ \textbf{csa\_mappings}(cr,csaid)}} \Big(\textbf{cs\_mp}_{opt}(dsaid, cdm, pred, pattrs) \times \textbf{cs\_nmp}_{opt}(cr, cdm, csaid, pattrs)\Big) \tag{1}$$

$$\textbf{csa\_mappings}(cr, csaid) = \pi_{\substack{csa.dsaid, \\ cs\_ds\_m.cr\_dr\_ms\rightarrow cdm}} \left(\sigma_{\substack{csa.id=csaid \\ \wedge cs.crs.name=cr}} \left(cs\_ds\_m \underset{cs\_ds\_m.id=csa.cs\_ds\_mid}{\bowtie} (\mu_{cs\_ds\_ms}(csa))\right)\right) \tag{2}$$

$$\textbf{cs\_mp}_{opt}(dsaid, cdm, pred, pattrs) = \pi_{id, \left\{[da]\rightarrow[ca] | (da,ca)\in \pi_{\substack{cdm.corrs.da, \\ cdm.corrs.ca}}(cdm) \wedge ([ca]\in pattrs \vee pattrs=NULL\right\}} \Big(\alpha_{opt}(cdm.dr, dsaid, \textbf{replace\_pred}(pred, cdm), \textbf{replace\_pattr}(pattrs, cdm))\Big) \tag{3}$$

$$\textbf{cs\_nmp}_{opt}(cr, cdm, csaid, pattrs) = \bigtimes_{\substack{ca\in\textbf{cattrs}(cr,csaid)\wedge \\ \textbf{is\_empty}(\sigma_{cdm.corrs.ca=[ca]}(cdm))\wedge \\ [ca]\in pattrs\vee pattrs=NULL)}} \underset{\rightarrow[ca](value,meta(mid,cid,type))}{(NULL, (NULL, NULL, NULL))} \tag{4}$$

$$\textbf{cattrs}(cr, csaid) = \underset{cs.crs.attrs}{\pi} \Big(\sigma_{\substack{csa.id=[csaid] \\ \wedge cs.crs.name=[cr]}} (cs \underset{cs.id=csa.csid}{\bowtie} csa)\Big) \tag{5}$$

$$\textbf{replace\_pred}(pred, cdm) = \textbf{string\_replace}(pred, ca, da) \forall (ca, da) \in \pi_{\substack{cdm.corrs.ca, \\ cdm.corrs.da}}(cdm) \tag{6}$$

$$\textbf{replace\_pattr}(pattrs, cdm) = \{da | (ca, da) \in \pi_{\substack{cdm.corrs.ca, \\ cdm.corrs.da}}(cdm) \wedge ca \in pattrs\} \tag{7}$$

Equation 3.7: Optimized Canonical Apply

$$\equiv \bigcup(\mathbf{cs\_mp}_{opt}(dsaid, cdm, \mathbf{true}, pattrs))) \bigtimes \pi_{pattrs}(\mathbf{cs\_nmp})) \quad (7)$$

by the definition of the $\mathbf{cs\_mp}_{opt}$ function (Function 3.7.3).

$$\equiv \bigcup(\mathbf{cs\_mp}_{opt}(dsaid, cdm, \mathbf{true}, pattrs))) \bigtimes \pi_{pattrs}(ca_1 \times \ldots \times ca_n)) \quad (8)$$

by the definition of the **cs_not_mapped** function (Function 3.2.4).

Here $ca_1, \ldots, ca_n$ are the canonical attributes not in the mapping cdm.

$$\equiv \bigcup(\mathbf{cs\_mp}_{opt}(dsaid, cdm, \mathbf{true}, pattrs))) \bigtimes (\times ca_j | ca_j \in pattrs)) \quad (9)$$

by the definition of the project operator [49]. Here $ca_j \in \{ca_1, \ldots, ca_n\}$ and we use $\times ca_j$to represent the cross-product of all such ca.

$$\equiv \theta_{opt}(cr, csaid, \mathbf{true}, pattrs) \quad (10)$$

by the definition of the *optimized canonical apply* operator (Equation 3.7).

□

We also provide an equivalence that allows selection predicates to be pushed into the *optimized canonical apply* operator. We assume that the predicate is well-formed and only contains literals or canonical attributes from the given cr.

Theorem 3.4. $\sigma_{pred}(\theta(cr, csaid)) \equiv \theta_{opt}(cr, csaid, pred, NULL)$

Proof.

$$\sigma \quad_{pred}(\theta(cr, csaid)) \equiv \sigma_{pred}(\bigcup(\mathbf{cs\_mp} \bigtimes \mathbf{cs\_nmp})) \quad (1)$$

by the definition of the *canonical apply* operator, Equation 3.2.

$$\equiv \bigcup(\sigma_{pred}(\mathbf{cs\_mp}) \bigtimes \mathbf{cs\_nmp}) \quad (2)$$

by the relational algebra equivalences $\sigma_\omega(E_0 \cup E_1) \equiv \sigma_\omega(E_0) \cup \sigma_\omega(E_1)$ and $\sigma_\omega(E_0 \times E_1) \equiv \sigma_\omega(E_0) \times \sigma_\omega(E_1)$ [1] and the requirement that only

mapped domain attributes and literals exist in *pred*.

$$\equiv \bigcup(\sigma_{pred}(\pi_{[da_i]\rightarrow[ca_i]}(\alpha(dr, dsaid))) \bigtimes \textbf{cs\_nmp}) \tag{3}$$

by the definition of the **cs_mapped** function (Function 3.2.3).

Here all ca_i are in the mapping passed to **cs_mapped**.

$$\equiv \bigcup(\sigma_{pred}(\pi_{[ca_i]}(\rho_{[da_i]\rightarrow[ca_i]}(\alpha(dr, dsaid)))) \bigtimes \textbf{cs\_nmp}) \tag{4}$$

by the definition of the project and renaming operators [49].

$$\equiv \bigcup(\pi_{[ca_i]}(\rho_{[da_i]\rightarrow[ca_i]}(\sigma_{pred'}(\alpha(dr, dsaid)))) \bigtimes \textbf{cs\_nmp}) \tag{5}$$

by the relational algebra equivalence $\sigma_\omega(E_0 \cup E_1) \equiv \sigma_\omega(E_0) \cup \sigma_\omega(E_1)$,

the associative relational algebra equivalence $\sigma_\omega(\pi_\delta(E_0)) \equiv \pi_\delta(\sigma_\omega(E_0))$

and the definition of the rename operator [49], which means that *pred* and

$pred'$ will be the same, where $pred' = \textbf{replace\_pred}(pred, cdm)$. (6)

$$\equiv \bigcup(\pi_{[ca_i]}(\rho_{[da_i]\rightarrow[ca_i]}(\alpha_{opt}(dr, dsaid, pred', NULL))) \bigtimes \textbf{cs\_nmp}) \tag{7}$$

by Theorem 3.2.

$$\equiv \theta_{opt}(cr, csaid, pred, NULL) \tag{8}$$

by the definition of the *optimized canonical apply* operator (Equation 3.7).

□

3.8.3 Removing Joins From Apply

If all local-relation-to-domain-relation mappings in a domain-structure application contain only correspondences where there are one or fewer correspondences to each domain attribute from unique local attributes, we can use a form of the *apply* operator that removes extraneous union and join operations. The *optimized apply*

without joins operator (Equation 3.8) is based on the *optimized apply* operator (Equation 3.6), so it can leverage previous optimizations.

The *optimized apply* operator is modified by replacing the **mp** function with a single instance of the $\mathbf{ptn}_{nj}$ function. The $\mathbf{ptn}_{nj}$ function (Function 3.8.3) combines all the project, select, and nest operations against a single **table_scan** operation. The project and nest operations are limited by the attributes in *pattr* and the select operator includes the passed in *pred* parameter, transformed to work against local attributes.

Using the *optimized apply without joins* operator when we are in the appropriate specific mapping case, we provide the following equivalence.

Theorem 3.5. $\alpha_{opt}(dr, dsaid, pred, pattrs) \equiv \alpha_{nj}(dr, dsaid, pred, pattrs)$ *if and only if the domain-structure application contains only mappings where each domain attribute exists in a single correspondence for each local-relation-to-domain-relation mapping.*

Proof.

$$\alpha_{opt} \qquad (dr, dsaid, pred, pattrs) \equiv \bigcup(\mathbf{mp}_{opt} \bigtimes \mathbf{nmp}_{opt}) \tag{1}$$

by the definition of the optimized apply operator (Equation 3.6).

$$\equiv \bigcup((\mathbf{ptn}_{opt}(da_1) \bowtie_{id} \ldots \bowtie_{id} \mathbf{ptn}_{opt}(da_n)) \bigtimes \mathbf{nmp}_{opt}) \tag{2}$$

by the definition of the $\mathbf{mapped}_{opt}$ function (Function 3.6.3).

$$\equiv \bigcup((\nu_{da_1}(\bigcup(\pi_{id,la_1 \to da_1}(lr))) \bowtie_{id} \ldots) \bigtimes \mathbf{nmp}_{opt}) \tag{3}$$

by the definition of the $\mathbf{proj\_type\_nest}_{opt}$ function (Function 3.6.4).

$$\equiv \bigcup((\nu_{da_1}(\pi_{id,la_1 \to da_1}(lr)) \bowtie_{id} \ldots) \bigtimes \mathbf{nmp}_{opt}) \tag{4}$$

by the requirement there is a unique correspondence per domain attribute.

$$\alpha_{nj}(dr, dsaid, pred, pattrs) = \bigcup_{\substack{(ldbid,dlm)\in \\ \textbf{dsa\_mappings}(dr,dsaid)}} \Big(\textbf{ptn}_{nj}(ldbid, dlm, pred, pattrs) \times \textbf{nmp}_{opt}(dr, dlm, dsaid, pattrs)\Big) \tag{1}$$

$$\textbf{dsa\_mappings}(dr, dsaid) = \pi_{\substack{ds\_ldb\_m.ldbid, \\ ds\_ldb\_m.dr\_lr\_ms\rightarrow dlm}} \left(\sigma_{\substack{ds\_ldb\_m.dr\_lr\_ms.dr=dr \\ \wedge dsa.id=dsaid}} \Big(ds\_ldb\_m \underset{ds\_ldb\_m.id=dsa.ds\_ldb\_mid}{\bowtie} (\mu_{ds\_ldb\_ms}(dsa))\Big)\right) \tag{2}$$

$$\textbf{ptn}_{nj}(ldbid, dlm, pred, pattrs) = \nu_{\substack{\{value_i, meta_i:[da_i]| \\ [corr_i.da]\in\pi_{dlm.corrs.da}(dlm)\wedge \\ ([corr_i.da]\in pattrs \vee pattrs=NULL)\}}} \Bigg(\tag{3}$$

$$\Big(\pi_{\substack{\textbf{gen\_key}(ldbid,dlm)\rightarrow id, \\ \{[corr_i.la]\rightarrow value_i| \\ [corr_i.da]\in\pi_{dlm.corrs.da}(dlm)\wedge \\ ([corr_i.da]\in pattrs \vee pattrs=NULL)\}}} (\sigma_{\substack{dlm.p\wedge \\ \textbf{replace}(pred,dlm)}} (\textbf{table\_scan}(ldbid, dlm.lr)))\Big)$$

$$\times \Big((dlm.id, corr_1.id, corr_1.la), \ldots, (dlm.id, corr_n.id, corr_n.la)\Big)\Bigg)$$
$$\underset{\rightarrow meta_1(mid,cid,type)}{\qquad\qquad} \underset{\rightarrow meta_n(mid,cid,type)}{\qquad\qquad}$$

$$\forall da_i \in \sigma_{da\in pattrs \vee pattrs=NULL}(\pi_{dlm.corrs.da}(dlm))$$

$$\textbf{gen\_key}(ldbid, dlm) = ldbid||.||dlm.lr||.||dlm.id||.||[\pi_{ldb.lrs.key}(\sigma_{\substack{ldb.lrs.name=dlm.lr \\ \wedge ldb.id=ldbid}}(ldb)] \tag{4}$$

$$\textbf{replace}(pred, dlm) = \textbf{string\_replace}(pred, da, la) \forall (da, la) \in \pi_{\substack{dlm.corrs.da, \\ dlm.corrs.la}}(dlm) \tag{5}$$

$$\textbf{nmp}_{opt}(dr, dlm, dsaid, pattrs) = \bigtimes_{\substack{da\in\textbf{dattrs}(dr,dsaid)\wedge \\ \textbf{is\_empty}(\sigma_{dlm.corrs.da=[da]}(dlm))\wedge \\ ([da]\in pattrs \vee pattrs=NULL)}} \underset{\rightarrow [da](value,meta(mid,cid,type))}{(NULL, (dlm.id, NULL, NULL))} \tag{6}$$

$$\textbf{dattrs}(dr, dsaid) = \pi_{ds.drs.attrs}\Big(\sigma_{\substack{ds.drs.name=[dr] \\ \wedge dsa.id=[dsaid]}}(ds \underset{ds.id=dsa.dsid}{\bowtie} dsa)\Big) \tag{7}$$

Equation 3.8: Optimized Apply without joins

$$\equiv \bigcup (\nu_{da_1,\overset{...,}{da_n}} (\pi_{id,la_1 \rightarrow da_1,\overset{...,}{la_n \rightarrow da_n}} (lr)) \bigtimes \mathbf{nmp}_{opt}) \quad (5)$$

since *id* is the key for the relation *lr* and no local attribute is repeated.

$$\equiv \alpha_{nj}(dr, dsaid, pred, pattrs) \quad (6)$$

by the *optimized apply without joins* operator (Equation 3.8).

□

3.9 PERFORMANCE ANALYSIS

Our system has been designed to facilitate non-technical users in performing mapping tasks and developers in using those mappings. If however, those features come at too great a cost, the system will not be used. Here, we evaluate the overhead imposed by our system from our extra layers of modeling and mappings.

We compare our system against a hard-coded custom widget that performs queries directly against its own schema and stores all data in a single table, requiring no joins in the resultant query. For the results in Table 3.9, this system is referred to as HC (hard-coded). Since the hard-coded system does not perform any of the overhead associated with our system, we consider the hard-coded system to be a good target for fast performance that we would hope to achieve in our best case. Our best-case scenario (USb) has only simple mappings and uses the *optimized apply without joins* operator described above.

We also compare ourselves to the default Drupal rendering system (labeled D in Table 3.9). Drupal stores each attribute of an entity in a separate database table, so, in order to render a page, it must create a join query joining all the tables of all the attributes. This is similar to our worst-case (USw) performance because if a user has composed complex mappings that involve multiple conditional correspondences mappings, our system performs a similar join query. Note also

Table 3.1: Performance comparison of our system in a best-case scenario (USb) and worst-case scenario (USw) to a hard-coded (HC) single query widget (an optimal but most labor intensive solution) and to the Drupal (D) page rendering system (a generic widget that can render arbitrarily complex types). All three systems tested with 2, 10, and 20 attributes. All times in milliseconds.

Rows	HC2	HC10	HC20	D2	D10	D20
100	6.2	7.2	8	6.6	29.6	47
1000	8.8	16.9	19.9	7.5	40.3	72.9
10000	31.5	79.1	129.6	40	145.7	326.5
Rows	USb2	USb10	USb20	USw2	USw10	USw20
100	6.5	9.9	12.6	7.3	33.5	52.6
1000	9.4	27.4	39.5	9.9	53.3	93.7
10000	46.9	174.5	322.9	67.9	245.3	524.8

that, like Drupal (and most other web systems), these costs are usually one-time costs, since the output of these queries can be cached.

Table 3.9 shows the results of the performance test. Our system is shown in both the best-case (USb) and worst-case (USw) scenarios. All systems were tested with 2, 10, and 20 attributes and on a database with 100, 1000, and 10000 entries. Times are shown in milliseconds and are the average of 10 runs each. All tests were performed on a server with an Intel I7 processor and 8GB of RAM.

From Table 3.9 we see that, in our best-case scenario, we are competitive to a hard-coded solution for a smaller number of rows, which is a great result for our naive implementation directly written against the formalism. This naive implementation introduces constants for mapping and type information for every attribute in every row which, unsurprisingly, leads to the slower performance at larger row and attribute sizes. Even with this overhead, we are comparable to

Drupal in our worst-case scenario and the same or better in our best-case, even at larger row sizes. Note that our system is performing local radiance, which is not done by either the hard-coded or Drupal system.

The test above compared systems using all attributes and data so the selection and projection optimizations of the *optimized apply* operator were not used. To test selection and projection optimizations, we used the same system as above and queried a domain relation with three domain attributes. The domain relation was mapped to twenty different local relations, each populated with 500,000 rows of data. We tested a selection operator with an equality predicate returning a single row. We also tested a projection operator that projected out a single domain attribute. Each query was run 100 times and the average of the times is shown in Table 3.2.

Table 3.2: Performance data for pushing projection and selection operators into the optimized apply operator.

	Unpushed	Pushed
Select	1,170.70ms	16.58ms
Project	1,168.73ms	37.11ms

In both cases we see a decrease of two orders of magnitude compared to the unoptimized operators as a baseline. The minimal overhead of having to check if an attribute list or predicate has been passed to the optimized operator can result in drastic performance improvement.

3.10 RELATED WORK

Our *apply* and *canonical apply* operators use a global-as-view model similar to traditional integration [48], but where traditional integration enforces a rigid singular global schema, we use many small global schema fragments (domain and canonical

structures). Our domain and canonical structures can also be seen as abstract superclasses of the various local schema types to which the domain structures have been mapped, similar to view integration and cooperation [79]. We extend these by bringing the local semantics through to the integrated functionality using our τ operator. The flexibility of our mappings and our operators' ability to handle incomplete to full mappings is also inspired by pay-as-you-go data integration, such as that proposed by Madhavan [29].

Bringing local schema metadata to a global integration has been studied and developed in systems such as SchemaSQL [47] and the Federated Interoperable Relational Algebra (FIRA) [87] and has been added to systems like Clio [40]. These systems address the problem that when integrating heterogeneous schemata it is often the case that data in one schema may exist as metadata in another schema (e.g., one schema may have city as an attribute of a company table whereas another schema may have one table for every city the company has an office in). Such systems often use the pivot and unpivot operation [78, 86] to transform schema into data (unpivot) or data into schema (pivot). In contrast, we bring local schema metadata to our domain and canonical structures in order to bring the local semantics to the global level through the use of the type operator. We also attempt to lower the complexity by performing local radiance by letting users add the local type operator to any domain or canonical relation at any point in a query by simply using another relational algebra operator. We believe using our operators is more intuitive than using database variables (in the case of SchemaSQL), having to deal with (possibly large) extraneous data as a result of the *down* operator in FIRA, or being limited solely to the attribute metadata in the case of pivot and unpivot. One trade-off of our lowered complexity is that we limit the possible mappings in our system, meaning that we have also lowered the possible transformations that can be expressed in our system.

As the usage of the semantic web [6] has grown, the number and variety of

schemata within it has also increased, requiring the introduction of integration concepts long known in databases. Ontologies have replaced global schemas [61] and traditional integration techniques have been used, but again, this type of information integration lacks flexibility. In contrast, other systems use small schemas (e.g. shallow or lightweight ontologies [75]) for search engines and other web integrations, such as those expressed in Microformats [55]. The use of Microformats requires that the schema elements are directly tied to the local data, making it difficult to compose different schemas and requiring editing the existing data to add global schema elements. These small schemas, as well as larger ontologies, have been used to create web widgets [53, 60] similar to our widgets, but they are limited to presenting the data in the form of the global schema, e.g., schema.org or an ontology, whereas our widgets can bring local semantics through.

3.11 SUMMARY

In this chapter we formally defined the *apply*, *canonical apply*, *apparent model*, and *type* operators. We presented examples cases for each operator and showed how the local, domain, and canonical structures and their associated mappings (presented in Chapter 2) can be used with the operators to provide a query system at the domain and canonical levels.

We presented optimizations to the *apply* and *canonical apply* operators and equivalences that allow the optimized versions to be used in conjunction with other relational equivalences to optimize queries in our system. Performance results show that the overhead added by our system is comparable to that of standard web content management systems and that using our optimized operators can speed up queries.

Chapter 4

BEYOND LOCAL RADIANCE TO LOCAL INSERT AND UPDATE

The previous chapters have shown how we facilitate the construction of generic widgets (such as a structured navigation menu) that work with all mapped elements of local databases in a system. Local radiance allows us to bring local type information from the various databases for use by the generic widgets.

Here, we consider how to enable generic widgets to insert and update local data—including local data that is not mapped to the global schema. The challenge then is how to generically modify data in the various local schemas using a domain structure[1] that is (by design) not complete. That is, how can we access local schema and data that sit outside the mappings?

In this chapter, we present extensions to our query language, originally defined in Chapter 3, that enable access to all local schema and data from the domain level (specifically to access the elements of the local relation that have not been mapped as long as at least one attribute of the local relation is mapped) as well as the ability to update and insert data locally from the domain level.

We make the following contributions:

- We define the *local document* operator (β) that, given a domain relation, will return a document for every tuple in the result of an apply operator on that domain relation. Each returned tuple contains the schema (from the local

[1]Note, as the work in this thesis has progressed we believed that we would only need domain structures and then later decided that we needed both domain and canonical structures. When the work of this chapter was performed we believed we only needed domain structures. This chapter therefore references domain structures as the end query model and not canonical structures.

relation) and data for all attributes from the local tuple that corresponds to the mapped tuple.

- We define the *empty document* operator (ϵ) that, given a domain relation, will return an empty document in the schema of each local relation that has been mapped to the domain relation.
- We define insert and update operators that use β and ϵ to insert and update local data from the domain level.
- We present a case study that demonstrates the use of the new operators.

4.1 LOCAL INSERT AND UPDATE

We introduce the four operators that enable global manipulation of local data in Table 4.1. The first two are the *local document* operator (β) and *empty document* operator (ϵ). Each of these operators, given a domain relation (dr) and a domain structure application ($dsaid$), will add an attribute to the query result containing a self-describing document that represents the full local schema of the elements in the local relation to which the domain relation has been mapped. The difference between the two operators is that β will generate a document populated with data from the local database whereas ϵ will generate an empty document with no data (but with the full local schema structure).

The *insert* and *update* operators allow local data creation and modification from the global level. Given a self-describing document (Doc)—such as those created by the β and ϵ operators, the *insert* operator ($InsertDocument(Doc)$) translates the document into the appropriate insert statement for the local database. The *update* operator ($UpdateDocument(Doc)$) translates a given document into the appropriate local update statement. Inserts or updates may fail if local schema constraints (e.g., not null or cardinality constraints) are not met.

Table 4.1: Extended query operators.

Operator	Name
$\beta(dr, dsaid)$	Local Document Operator
$\epsilon(dr, dsaid)$	Empty Document Operator
$InsertDocument(Doc)$	Insert Document Operator
$UpdateDocument(Doc)$	Update Document Operator

The limitation of our mapping system—that a domain relation may only be mapped to a single local relation and not a join of multiple local relations—allows us to avoid the view update problem of updating over a join path. Since the local documents we produce contain the full schema of the local relations, not just the parts mapped to domain structures, we need not worry about the view update problem due to projection.

Local Document Operator

Given a domain relation (dr) and a domain structure application identifier ($dsaid$) the local document operator (β) is defined as shown below in Equation 4.1. The local document operator uses the operators and structures within our system defined in Chapters 2 and 3, repeated below.

Relations for local and domain structures and mappings.

Local DBs:

$ldb(\underline{id}, lrs(name, key, attrs(name)))$

Domain Structures:

$ds(\underline{id}, drs(name, key, attrs(name)))$

Domain Structure - Local DB Mappings:

$ds\_ldb\_m(\underline{id}, ldbid, dsid, dr\_lr\_ms(id, lr, dr, p, corrs(id, la, da)))$

Domain Structure Application:

$dsa(\underline{id}, dsid, ds\_ldb\_ms(ds\_ldb\_mid))$

Also, recall the type of the result from the apply operator, α, is:

$name(id, attr_1(value, meta(mid, cid, type)), \ldots,$

$attr_n(value, meta(mid, cid, type)))$

where $name \in \pi_{ds.drs.name}(ds)$ and every

$attr_i \in \pi_{ds.drs.attrs.name}(\sigma_{ds.drs.name=name}(ds))$ appears in this expression.

We provide an example to help explain how the *local document* operator functions. For this example, we continue with the sports databases from the previous chapters; in particular we will focus on the local relations shown below.

Result Set 4.1

```
# select * from ldb;
    id     |                  lrs(name,key,attrs(name))
-----------+-------------------------------------------------------
FootballDB | (Employee,EmployeeId,"{EmployeeId,EmployeeName,Address}")
TennisDB   | (Student,StudentId,"{StudentId,Name,gpa}")
```

Here, the football database has a single local relation ("Employee") that has three local attributes ("EmployeeId", "EmployeeName", and "Address"). The tennis database has a single local relation ("Student") that has three local attributes ("StudentId", "Name", and "gpa"). Example local data for these two databases is shown below.

Result Set 4.2

```
# select * from tennisdb.student;
studentid | name  | gpa
----------+-------+-----
1         | Alice | 4.0
2         | Bob   | 3.5

# select * from footballdb.employee;
employeeid | employeename |    address
-----------+--------------+---------------
999        | Sue          | 123 Main St.
1001       | John         | 34 Union Ave.
```

For this example, we use the following domain structure ("TeamDS") that has a single domain relation, ("Person") that has two domain attributes ("PersonId" and "GivenName")

Result Set 4.3

```
# select * from ds;
    id     |             drs(name,key,attrs(name))
-----------+-----------------------------------------------------
TeamDS     | (Person,id,{PersonId,GivenName})
```

$$\beta(dr, dsaid) = \bigcup_{mid \in \textbf{mids}(dr, dsaid)} \textbf{build\_local}(mid) \tag{1}$$

$$\textbf{mids}(dr, dsaid) = \pi_{\textbf{SPLIT}(id,\text{`.'})[2]}(\alpha(dr, dsaid)) \tag{2}$$

$$\textbf{build\_local}(mid) = \nu_{mid,ldb,lr,lrkey,attrs:local\_doc}\left(\nu_{lrkeyattr,lrkeyvalvalue:lrkey}\left(\nu_{name,value:attrs}\left(\bigcup_{\substack{(ldbid,lr,lrkey,attr)\in \\ \textbf{local\_from\_mid}(mid)}} \left(\pi_{\substack{\text{“}[ldbid]||.||[lr]||.||[mid]||.||[lrkey]\text{''}\rightarrow id, \\ \text{“}[mid]\text{''}\rightarrow mid, \text{“}[ldbid]\text{''}\rightarrow ldb, \text{“}[lr]\text{''}\rightarrow lr, \\ \text{“}[lrkey]\text{''}\rightarrow lrkeyattr, [lrkey]\rightarrow lrkeyval \\ \text{“}[attr]\text{''}\rightarrow name, \\ [attr]\rightarrow value}} (\textbf{table\_scan}(ldbid, lr))\right)\right)\right)\right) \tag{3}$$

$$\textbf{local\_from\_mid}(mid) = \pi_{\substack{ldb.id\rightarrow ldbid, \\ ldb.lrs.name\rightarrow lr, \\ ldb.lrs.key\rightarrow lrkey, \\ ldb.lrs.attrs.name\rightarrow attr}} \left((\sigma_{ds\_ldb\_m.dr\_lr\_ms.id=mid}(ds\_ldb\_m)) \bowtie_{\substack{ds\_ldb\_m.ldbid=ldb.id \wedge \\ ds\_ldb\_m.dr\_lr\_ms.lr=ldb.lrs.name}} ldb\right) \tag{4}$$

Where the **table_scan**($ldbid, lr$) function performs a table scan operation on the local relation lr in the local database $ldbid$; and, the **SPLIT**(s,d) function splits a string (s) on a delimiter (d) and returns an array of the resulting substrings.

Equation 4.1: *Local Document* Operator (β)

Mappings between the local databases and the domain structure are defined below.

Result Set 4.4

```
# select * from ds_ldb_m;
 id |   ldbid    |  dsid  |         dr_lr_ms(id,lr,dr,p,corrs(id,la,da)
----+------------+--------+-------------------------------------------------------------
 2  | FootballDB | TeamDS | (70,Employee,Person,TRUE,{(700,EmployeeId,PersonId),
                                                        (701,EmployeeName,GivenName)})
 1  | TennisDB   | TeamDS | (30,Student,Person,TRUE,{(300,StudentId,PersonId),
                                                      (301,Name,GivenName)})
```

The local football database has a mapping between the "Employee" local relation and the "Person" domain relation with two correspondences (between the "EmployeeId" local attribute and "PersonId" domain attribute and between the "EmployeeName" local attribute and "GivenName" domain attribute). The local tennis database has a mapping between the "Student" local relation and the "Person" domain relation with two correspondences (between the "StudentId" local attribute and "PersonId" domain attribute and between the "Name" local attribute and "GivenName" domain attribute).

Given these structures and mappings we will show how each step of the *local document* operator ($\beta(Person, 1)$) works, where 1 is the id of the domain structure application that contains the two mappings described above and shown below.

Result Set 4.5

```
# select * from dsa;
id |  dsid  | ds_ldb_mid
---+--------+------------
1  | TeamDS | {1,2}
```

To start building the local documents for each tuple of the domain relation, we first find all the mapping ids using the **mids**$(dr, dsaid)$ function (Function 4.1.2). This functions runs the *apply* operator on the domain relation and then extracts all the mapping ids from the id attribute of the domain relation. The *apply* operator, α, acting upon this instance of the two local databases results in the following relation.

Result Set 4.6

```
# select * from alpha('Person','1')
             id             |            PersonId          |             GivenName
----------------------------+------------------------------+-----------------------------
TennisDB.Student.30.2       |{(2,(30,300,StudentId))}      |{(Bob,(30,301,Name))}
TennisDB.Student.30.1       |{(1,(30,300,StudentId))}      |{(Alice,(30,301,Name))}
FootballDB.Employee.70.1001|{(1001,(70,700,EmployeeId))}|{(John,(70,701,EmployeeName))}
FootballDB.Employee.70.999 |{(999,(70,700,EmployeeId))} |{(Sue,(70,701,EmployeeName))}
```

The **mids** function (Function 4.1.2) then extracts the mapping ids from the "id" attribute using a string split function to retrieve the third element of the dot delimited string of the generated id, in this case "30" and "70".

Result Set 4.7

```
# select * from mids('Person','1');
mid
-----
30
70
```

Based on the mapping ids retrieved, the β operator retrieves all the local attributes for each local relation ("Student" and "Employee") in the mappings referenced by the mapping ids ("30" and "70"), and then builds a nested relation that includes all the local relation attributes and values using the **build_local** function (Function 4.1.3).

At a high level, the **build_local** function gets the set of all local attributes in the local relation mapped in the given mapping (*mid*) (from the **local_from_mid** function), projects each individual attribute from the local relations as "name"-"value" pairs, unions all the attribute pairs and then nests the results into a local document that contains the mapping id, the local database name, the local relation name, the local relation key, and a nested relation of all local attributes with their names and values. We discuss this function in detail below.

The **build_local** function first uses the **local_from_mid** function (Function 4.1.4) to find all local attributes of the local relation (whether or not they have been mapped). Using the mapping information from *ds_ldb_m* (shown in Result Set 4) and the local database information from *ldb* (shown in Result Set 1), the **local_from_mid** function projects the local database id, the local relation name, the local relation-key name, and each attribute in the local relation. Here we use the nested-relational version of project, which unnests the $ldb.lrs.attrs$ nested relation and produces one tuple of output for each nested tuple in $ldb.lrs.attrs$. The results of this function for the first mapping id ("30") are shown below.

Result Set 4.8

```
# select * from local_info_from_mid('30');
 ldbid   |   lr    |   lrkey   |   attr
---------+---------+-----------+-----------
TennisDB | Student | StudentId | StudentId
TennisDB | Student | StudentId | Name
TennisDB | Student | StudentId | gpa
```

For each local attribute from the **local_from_mid** function the **build_local** function projects the domain relation id attribute, mapping id, local database id, local relation name, local relation key attribute name, local relation key value, and the local attribute name and value which are then combined in the union operation. The domain relation id is generated by the *apply* operator and includes the local database, local relation, mapping id, and key value which will make the result of the operator joinable with the results of an *apply* or *canonical apply* operation. This part of the function for the mapping id "30" produces the relational result below.

Result Set 4.9

```
         id           | mid| ldbid    |   lr    | lrkeyattr | lrkeyval | attr      | value
----------------------+----+----------+---------+-----------+----------+-----------+--------
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 1        | StudentId | 1
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 1        | Name      | Alice
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 1        | gpa       | 4.0
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 2        | StudentId | 2
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 2        | Name      | Bob
TennisDB.Student.30.1 | 30 | TennisDB | Student | StudentID | 2        | gpa       | 3.5
```

The "name" and "value" for each id are then nested into the "attrs" nested relation, which produces the following nested relational result:

Result Set 4.10

```
          id          |mid|  ldbid  |  lr   |lrkeyattr|lrkv|         attrs(attr,value)
----------------------+---+---------+-------+---------+----+-------------------------------------
TennisDB.Student.30.1| 30|TennisDB|Student|StudentID| 1  |{(StudentId,1),(Name,Alice),(gpa,4.0)}
TennisDB.Student.30.2| 30|TennisDB|Student|StudentID| 2  |{(StudentId,2),(Name,Bob),(gpa,3.5)}
```

The local relation key attribute name and value are nested in the "lrkey" nested relation as shown below.

Result Set 4.11

```
         id           |mid| ldbid  |  lr   |lrkey(lrka,lrkv)|        attrs(attr,value)
----------------------+---+--------+-------+----------------+-------------------------------------
TennisDB.Student.30.1| 30|TennisDB|Student| {StudentId,1}  |{(StudentId,1),(Name,Alice),(gpa,4.0)}
TennisDB.Student.30.2| 30|TennisDB|Student| {StudentID,2}  |{(StudentId,2),(Name,Bob),(gpa,3.5)}
```

The mapping id and local database id are then nested with the "lrkey" and "attrs" relations to create the local document. The **build_local** function (Function 4.1.3) for mapping id "30" will then return the following result:

Result Set 4.12

```
# select * from build_local('30');
          id           | local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
-----------------------+-------------------------------------------------------------------------
 TennisDB.Student.30.1 |{30,TennisDB,Student,{StudentId,1},{(StudentId,1),(Name,Alice),(gpa,4.0)}}
 TennisDB.Student.30.2 |{30,TennisDB,Student,{StudentId,2},{(StudentId,2),(Name,Bob),(gpa,3.5)}}
```

In a similar fashion, for the mapping above with id "70" the **local_from_mid** function (Function 4.1.4) will then return the local attributes as follows:

Result Set 4.13

```
# select * from local_from_mid('70');
  ldbid    |    lr    |   lrkey    |     attr
-----------+----------+------------+--------------
FootballDB | Employee | EmployeeId | EmployeeId
FootballDB | Employee | EmployeeId | EmployeeName
FootballDB | Employee | EmployeeId | Address
```

The **build_local** function (Function 4.1.3) for mapping id "70" will then return the following tuples:

Result Set 4.14

```
# select * from build_local('70');
             id            |local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
---------------------------+-------------------------------------------------------------------------
FootballDB.Employee.70.1001|{70,FootballDB,Employee,{EmployeeId,1001},{(EmployeeId,1001),
                                                                        (EmployeeName,John),
                                                                        (Address,34 Union Ave.)}}
FootballDB.Employee.70.999 |{70,FootballDB,Employee,{EmployeeId,999},{(EmployeeId,999),
                                                                      (EmployeeName,Sue),
                                                                      (Address,123 Main St.)}}
```

The *local document* operator (β) (Function 4.1.1) will then return the union of the results of the **build_local** function (Function 4.1.3) for all returned mapping ids as shown below.

Result Set 4.15

```
# select * from beta('Person','1');
          id               |local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
---------------------------+-------------------------------------------------------------------------
TennisDB.Student.30.1      |{30,TennisDB,Student,{StudentId,1},{(StudentId,1),(Name,Alice),
                                                                (gpa,4.0)}}
TennisDB.Student.30.2      |{30,TennisDB,Student,{StudentId,2},{(StudentId,2),(Name,Bob),
                                                                (gpa,3.5)}}
FootballDB.Employee.70.1001|{70,FootballDB,Employee,{EmployeeId,1001},{(EmployeeId,1001),
                                                                        (EmployeeName,John),
                                                                        (Address,34 Union Ave.)}}
FootballDB.Employee.70.999 |{70,FootballDB,Employee,{EmployeeId,999},{(EmployeeId,999),
                                                                      (EmployeeName,Sue),
                                                                      (Address,123 Main St.)}}
```

Empty Document Operator

Given a domain relation (*dr*) and a domain structure application id (*dsaid*), the *empty document* operator is defined as shown in Equation 4.2. The *empty document*

operator is similar to the *local document* operator and uses the same **mids** and **local_from_mid** functions as the *local document* operator. The only difference is in the **build_empty_local** function (Function 4.2.3). Continuing with the example above, the *empty document* operator will produce all the same results up through Result Set 7. Before nesting, the **build_empty_local** function will produce the following result for all attribtues returned from the **local_from_mid** function for mapping "30":

Result Set 4.16

```
 id  | mid| ldbid    |   lr    | lrkeyattr | lrkeyval | attr      | value
-----+----+----------+---------+-----------+----------+-----------+--------
NULL | 30 | TennisDB | Student | StudentID | NULL     | StudentId | NULL
NULL | 30 | TennisDB | Student | StudentID | NULL     | Name      | NULL
NULL | 30 | TennisDB | Student | StudentID | NULL     | gpa       | NULL
```

The **build_empty_local** function then builds a local document by nesting the above result in the same process as used above in Result Sets 9 and 10 to produce the following:

Result Set 4.17

```
# select * from build_empty_local('30');
 id  |   local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
-----+-------------------------------------------------------------------------
NULL | {30,TennisDB,Student,{StudentId,NULL},{(StudentId,NULL),(Name,NULL),(gpa,NULL)}}
```

The results of the **build_empty_local** function for mapping id "70" is then shown below.

Result Set 4.18

```
# select * from build_empty_local('70');
 id  |           local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
-----+-----------------------------------------------------------------------------
NULL | {70,FootballDB,Employee,{EmployeeId,NULL},{(EmployeeId,NULL),(EmployeeName,NULL),
                                                  (Address,NULL)}}
```

The *empty document* operator (ϵ) (Function 4.2.1) then returns the union of the results of the **build_empty_local** function for each mapping id as shown below.

Result Set 4.19

```
# select * from epsilon('Person','1');
 id  |           local_doc(mid,ldbid,lr,lrkey(lrkeyattr,lrkeyvalue),attrs(attr,value))
-----+-----------------------------------------------------------------------------
NULL | {30,TennisDB,Student,{StudentId,NULL},{(StudentId,NULL),(Name,NULL),(gpa,NULL)}}
NULL | {70,FootballDB,Employee,{EmployeeId,NULL},{(EmployeeId,NULL),(EmployeeName,NULL),
                                                  (Address,NULL)}}
```

In the next two subsections we show how local documents and empty documents are used for update and insert operations, respectively.

$$\epsilon(dr, dsaid) = \bigcup_{mid \in \mathbf{mids}(dr,dsaid)} \mathbf{build\_empty\_local}(mid) \tag{1}$$

$$\mathbf{mids}(dr, dsaid) = \pi_{\mathbf{SPLIT}(id,`.`)[2]}(\alpha(dr, dsaid)) \tag{2}$$

$$\mathbf{build\_empty\_local}(mid) = \nu_{ldb,lr,attrs:local\_doc}\Bigg(\nu_{lrkeyattr,lrkeyvalvalue:lrkey}\Bigg(\nu_{name,value:attrs}\Bigg(\bigcup_{\substack{(ldbid,lr,lrkey,attr)\in \\ \mathbf{local\_from\_mid}(mid)}} \underset{\rightarrow(id,mid,ldb,lr,lrkeyattr,lrkeyval,name,value)}{(NULL, [mid], [ldbid], [lr], [lrkey], NULL, [attr], NULL)}\Bigg)\Bigg)\Bigg) \tag{3}$$

$$\mathbf{local\_from\_mid}(mid) = \pi_{\substack{ldb.id\rightarrow ldbid, \\ ldb.lrs.name\rightarrow lr, \\ ldb.lrs.key\rightarrow lrkey, \\ ldb.lrs.attrs.name\rightarrow attr}} \Bigg((\sigma_{ds\_ldb\_m.dr\_lr\_ms.id=mid}(ds\_ldb\_m)) \underset{\substack{ds\_ldb\_m.ldbid=ldb.id \wedge \\ ds\_ldb\_m.dr\_lr\_ms.lr=ldb.lrs.name}}{\bowtie} ldb\Bigg) \tag{4}$$

Where the **table_scan**$(ldbid, lr)$ function performs a table scan operation on the local relation lr in the local database $ldbid$; and, the **SPLIT**(s,d) function splits a string (s) on a delimiter (d) and returns an array of the resulting substrings.

Equation 4.2: *Empty Document* Operator (ϵ)

4.1.1 Update

If the data of an existing local tuple has been changed in a local document, those changes can be propagated to the local database using the *update document* operator (defined in Algorithm 4.3 below).

Algorithm 4.3 Algorithm for building local updates from a local document.

```
1: procedure UpdateDoc(local_doc)
2:     updates ← array()
3:     for all (attr, value) in local_doc.attrs do
4:         sets.append(attr||'='||value)
5:     end for
6:     EXECUTE UPDATE local_doc.ldb.local_doc.lr SET updates.join(,)
   WHERE local_doc.lrkey.lrkeyattr = local_doc.lrkey.lrkeyval
7: end procedure
```

The *update document* operator executes an SQL update statement using the attribute names and values in the local document provided. An empty array is created to store strings of the form "attr = value". The operator then returns an empty statement by joining the update strings delimited by commas and only updating the record referenced by the local key in the local document. In order to avoid determining which attributes may have been updated, we take the straightforward approach of updating all local attributes.

So, for example, say the document for "Sue" above in the insert section was updated with a new gpa as follows:

```
{30,TennisDB,Student,{StudentId,'3'},{(StudentId,'3'),(Name,'Sue'),(gpa,'4')}}
```

The update document operator would first create the "updates" array:

```
updates = {StudentId='3',Name='Sue',gpa='4'}
```

An update statement is then produced using the local relation referenced in the local document (*local_doc.ldb.local_doc.lr*) and limited to the record in the document using the "WHERE" clause and the local key from the document

(*local_doc.lrkey.lrkeyattr* = *local_doc.lrkey.lrkeyval*). The algorithm then produces the following update statement:

```
UPDATE TennisDB.Student SET StudentId='3',Name='Sue',gpa='4' WHERE StudentId='3'
```

4.1.2 Insert

Once a generic widget has populated an empty document, we provide an operator for inserting that data into the appropriate local database. Given a local document *local_doc* our system will run the *insert document* operator, described below in Algorithm 4.4, on the local database referenced in the local document (*local_doc.ldb*).

Algorithm 4.4 Algorithm for building local inserts from a local document.

```
1: procedure InsertDoc(local_doc)
2:     attrs ← array()
3:     values ← array()
4:     for all (attr, value) in local_doc.attrs do
5:         attrs.append(attr)
6:         values.append(value)
7:     end for
8:     EXECUTE INSERT INTO local_doc.ldb.local_doc.lr (attrs.join(,)) VALUES (values.join(,))
9:     return local_doc.lrkey.lrkeyval
10: end procedure
```

The *insert document* operator builds an SQL insert statement based on the attribute names and values in the local document. First, two empty arrays are created for the attribute names and values (lines 2 and 3) and are populated with the data from the attributes in the local document (*local_doc.attrs*). The *join* function converts the arrays into comma-delimited strings, producing an insert statement of the following form:

```
INSERT INTO local_relation (attr1,...,attrn) VALUES (value1,...,valuen)
```

As an example, consider the local document created from the mapping to the tennis database above in Result Set 4.19.

```
{30,TennisDB,Student,{StudentId,NULL},{(StudentId,NULL),(Name,NULL),(gpa,NULL)}}
```

If a widget populates this document as follows:

```
{30,TennisDB,Student,{StudentId,'3'},{(StudentId,'3'),(Name,'Sue'),(gpa,'3.5')}}
```

The *insert document* operator for this document will then populate the two arrays:

```
attrs = {StudentId,Name,gpa}
values = {'3','Sue','3.5'}
```

The operator will then execute the following insert statement by performing the join operation on the two arrays:

```
INSERT INTO TennisDB.Student (StudentId,Name,gpa) VALUES ('3','Sue','3.5');
```

The local relation key value is then returned, which allows auto-generated key values to be known at the domain level if they were not populated in the local document.

4.2 CASE STUDY STEMROBOTICS

We demonstrate the use of all of the document operators in two widgets in the STEMRobotics[2] repository of educational materials. The repository contains schemas for different course structures, as well as books and other educational materials. One course schema is shown in Figure 4.1, where a course has units and a unit has lessons. A book schema (shown in Figure 4.2) has a book that has chapters and a chapter has sections.

The repository hosts master curricula that have been created to help teachers who are new to a subject. These curricula are often used alongside a professional development program where new teachers spend a week or two learning the materials in order to be ready to teach students. After using the materials as-is once or twice, teachers may find that they prefer to use the materials in a different order,

[2]http://stemrobotics.cs.pdx.edu

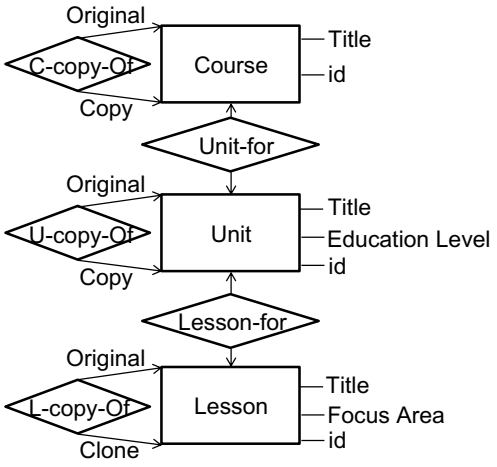


Figure 4.1: The course local schema.

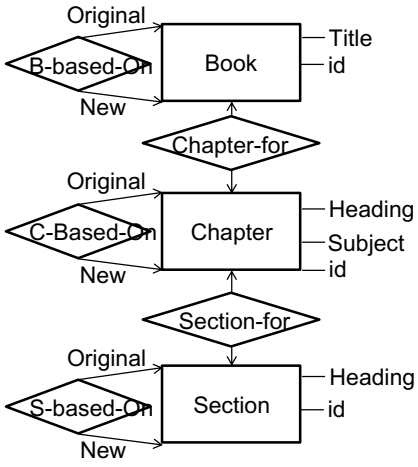


Figure 4.2: The book local schema.

add additional materials, or omit some materials. It is useful for them to create their own copy of the course, which they then modify to suit their specific needs. These same actions may also happen with a book. Our local schemas shown in Figures 4.1 and 4.2 include "X-copy of" and "X-based on" relationships, respectively, to track these copy and modify actions.

As an example, in the repository, the "STEM Robotics 101" course has been taught in numerous professional development programs. It is used by teachers throughout the United States in middle and high school classrooms as well as after-school programs. In many cases, a teacher has decided to rearrange and augment the master curriculum. To facilitate the teachers, we created the drag-and-drop cloning widget shown in Figure 4.3. The left side of the figure shows a clone of the "STEM Robotics 101" course being created. The user has selected the course guide, the classroom resources, all of unit 3, and lessons 1, 2, and 6 from unit 1. The user also moved unit 3 to come before unit 1. The right side of Figure 4.3 shows a similar process occurring for a book.

The repository supports cloning generically, across heterogeneous local

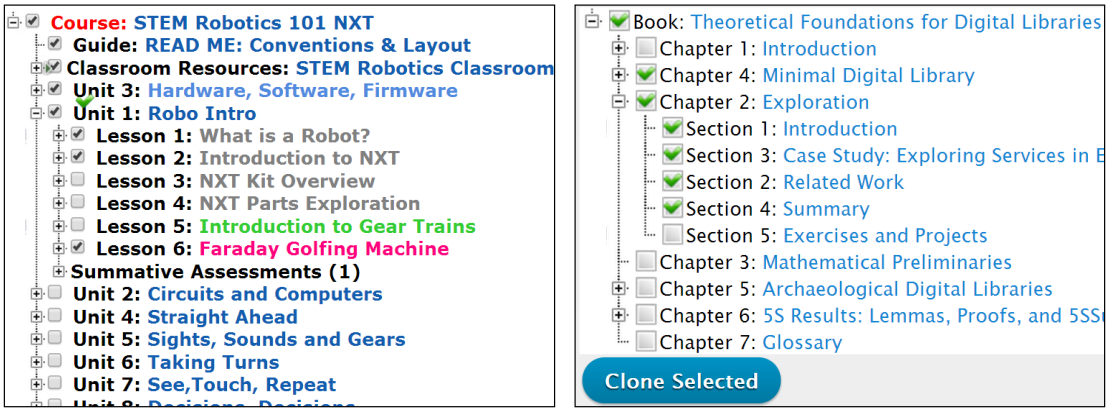


Figure 4.3: Widgets for cloning a course (left) and a book (right).

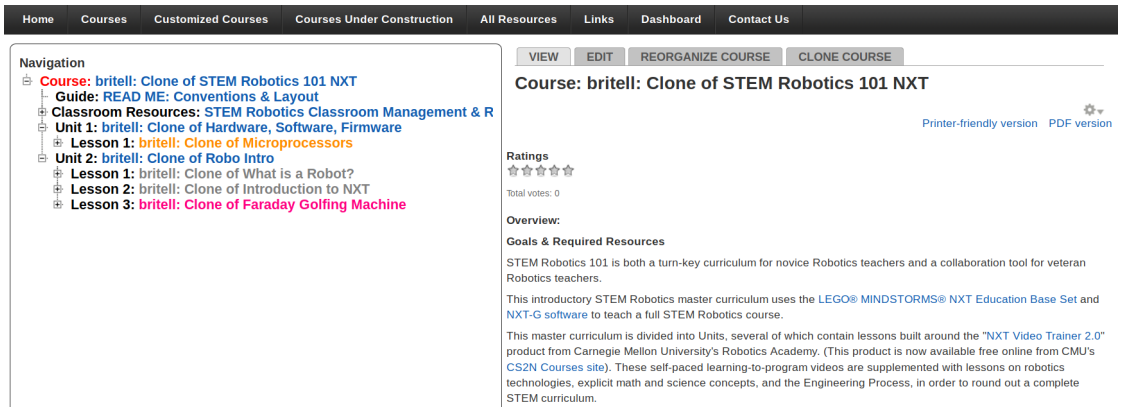


Figure 4.4: The cloned course page created by the course cloning widget on the left side of Figure 4.3.

schemas. Cloning requires creating new local content (such as populating "clone-of" relationships in the local schema and creating the new "cloned" items) so the global functionality must be able to perform local inserts and updates. And cloning requires the use of potentially all local fields—not just those that were mapped—in order to create the tree structured widget. This cloning widget and the following exploration widget will both be described in further detail below.

Seeing how the master course has been cloned and modified is important to the original author, because new materials added to clones can be valuable for other teachers using the original course. Additionally, since the master course is not static, it is useful for a teacher of the cloned course to see the differences between the master course and the clone. To view these modifications, we have developed a clone exploration widget, shown in Figure 4.5, to show the structural differences between a clone and the original. In the figure, a unit and its clone are being compared. The squares on the right represent the original unit and its lessons, while the squares on the left represent the cloned resources. Lines between the squares represent "Part Of" relationships (vertically oriented) and "Clone of" relationships (horizontally oriented). Additionally the widget can show if resources in the clone have been reordered compared to the originals. Here, all but one of the lessons have been cloned and one lesson is used as it is in the original in the clone (the small square in the middle of the figure linked to both units).

The generic clone exploration widget is written against a global schema. But in order to accurately compare the clones and the originals, we must extract everything about the local records, not just what is available through the mappings to the global schema.

4.2.1 Domain Structures Used in the Cloning and Exploration Widgets

To build the hierarchy used in the clone widget (Figure 4.3) we use the "Parent-Part" domain structure shown in Figure 4.6. The domain structure contains two

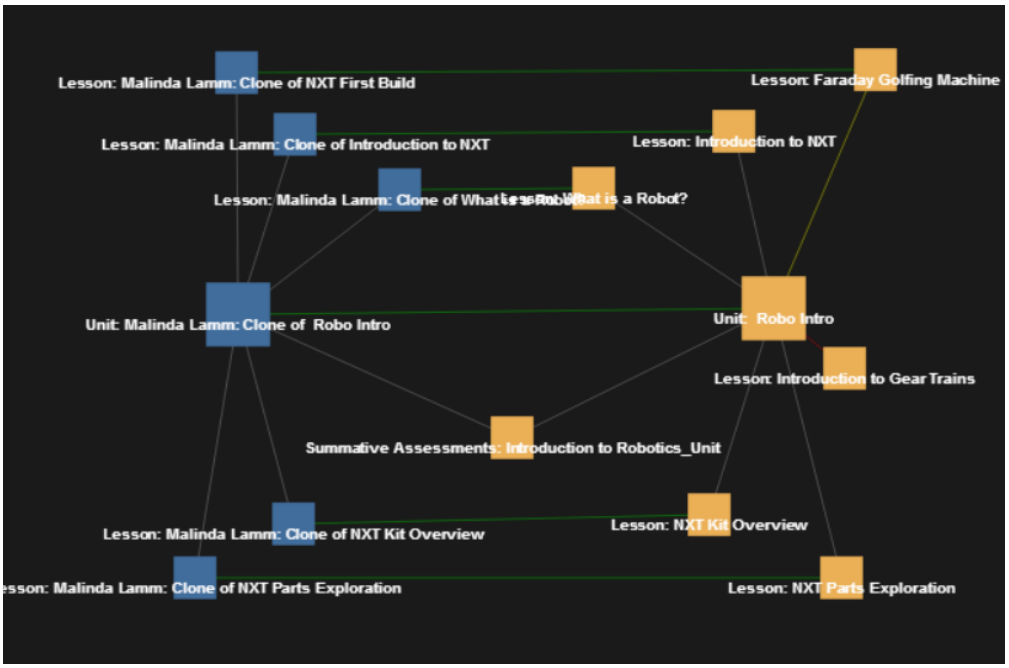


Figure 4.5: Exploring a clone of a course.

domain entities ("Parent" and "Part"), and a domain relationship ("Has"); each domain entity has an "id" and a "title" domain attribute. To populate local clone relationships and to enable the clone exploration widget, we use the clone domain structure shown in Figure 4.7. This structure consists of a single domain entity ("Thing") with "title" and "id" domain attributes, and the "Clone-Of" domain relationship with labels "Original" and "Clone" for the two ends of the domain relationship.

4.2.2 Mappings Used in the Cloning and Exploration Widgets

Figure 4.8 shows a set of mappings, drawn at the entity-relationship level, of the "Parent-Part" domain structure to the course schema. Here, correspondences have been drawn between the "Parent" domain entity and the "Course" local entity, as well as their respective "title" attributes. The "Has" domain relationship corresponds to the "Unit-for" local relationship. Correspondences have been drawn

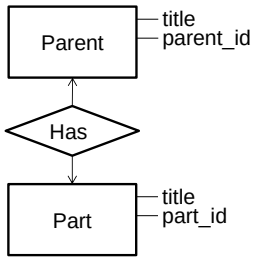


Figure 4.6: The Parent-Part domain structure.

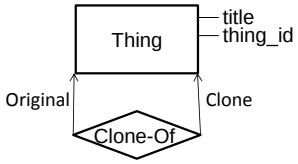


Figure 4.7: The clone domain structure.

from the "Part" domain entity and its "title" domain attribute to the "Unit" local entity and its "title" attribute.

Figure 4.9 shows a similar set of mappings of the "Parent-Part" domain structure to the book local schema. Correspondences have been drawn from the "Parent" domain entity to the "Chapter" local entity, from the "Has" domain relationship to the "Section-For" local entity, and from the "Part" domain entity to the "Section" local entity.

To build the entire hierarchy shown in the clone widget, the "Parent-Part" domain structure is mapped to all levels of the course and book schemas. Figures showing these mappings have been omitted for the sake of brevity.

Figures 4.10 and 4.11 show mappings of the "Clone-Of" domain structure to the course and book local schemas: the "Thing" domain entity is mapped to a local entity and the "Clone-Of" relationship is mapped to the local relationship attached to the mapped local entity.

4.2.3 Widgets

We have implemented our mapping and query interface using the Drupal [33] content management system. Widgets are added to Drupal by writing modules that can be enabled in a given site. Queries are written in our extended algebra and, when posed to the query interface, return a database result object identical

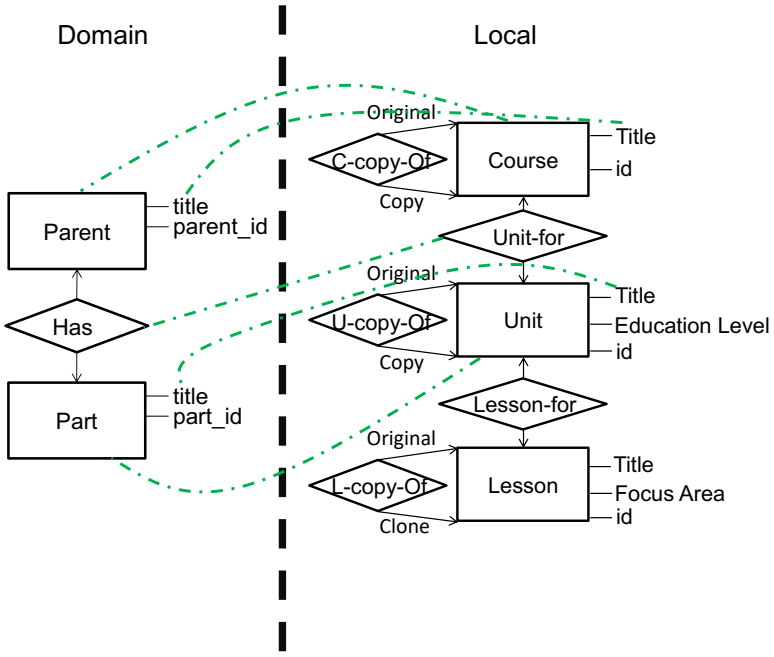


Figure 4.8: One mapping of the Parent-Part DS to the course schema

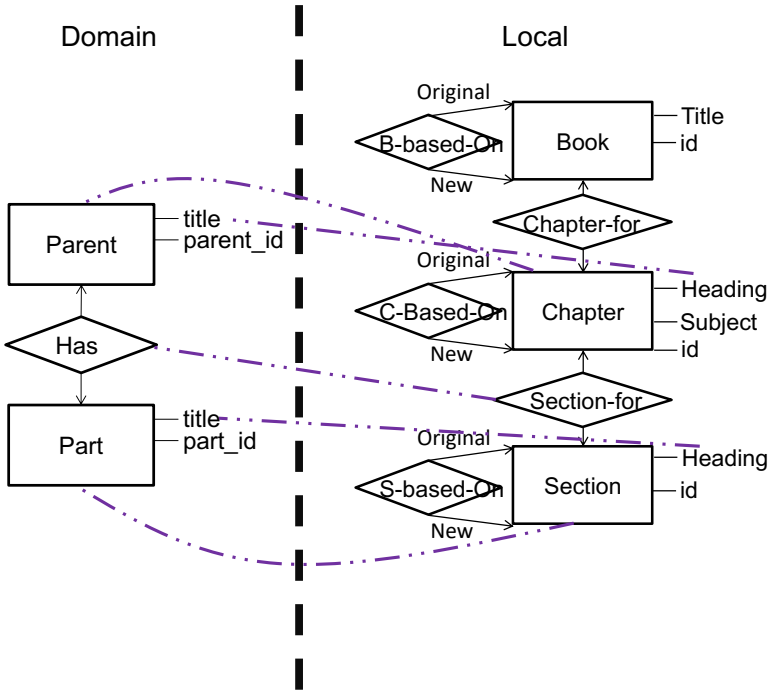


Figure 4.9: One mapping of the Parent-Part DS to the book schema

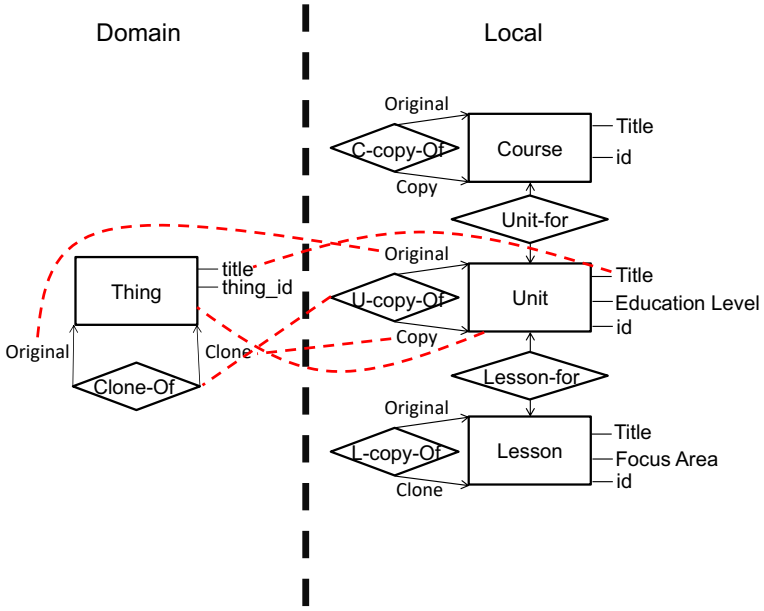


Figure 4.10: One mapping of the CloneOf DS to the course schema

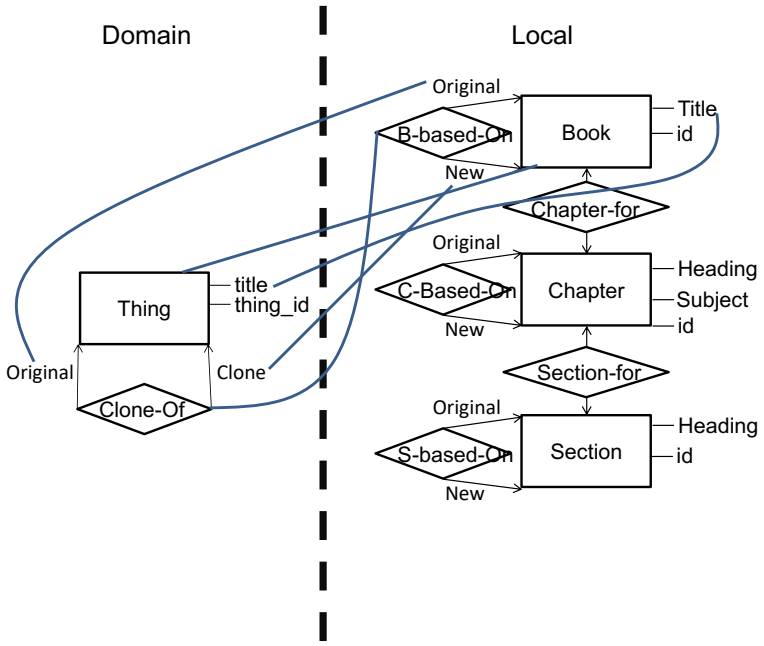


Figure 4.11: One mapping of the CloneOf DS to the book schema

in structure to the original database query objects defined in the base Drupal system. Using the query interface, developers can then write widgets as they would normally in Drupal, but gain the benefit of our integrated queries.

The Clone and Structural Edit Widget

The cloning widget shown in Figure 4.3 can both clone hierarchically structured data and rearrange existing data (what we call *structural editing*).

The cloning widget first issues a query against the "Has" domain structure, described above, that recursively builds the course tree by querying for the children of each level of the tree. The widget then uses the *local document* operator to retrieve all the local data associated with each level of the tree so that each clone has all local attributes. Figure 4.12 shows the course level page of the "STEM Robotics 101" course. The only part of this course page that is accessible from the "Parent" domain relation is the title, but if we wish to create a clone of the course (Figure 4.13) we need to also access the "Overview" attribute, for example. Or, when cloning the "Hardware, Software, Firmware" unit (Figure 4.14), the new clone (Figure 4.15) should contain all the data from the "Overview", "Education Level", "Focus Area", "HW Platform", "SW Platform", and "Interactivity Style" attributes[3].

As well as creating clones of the resources selected in the interface (Figure 4.3), the widget must populate the local "Unit-For" and "Lesson-For" relations in order to store the hierarchy of the clone of the course. We retrieve the local documents for the hierarchy of the existing course (using the "Has" domain relation) and then also retrieve empty documents that will be populated with the new clone ids based on the existing course. The new hierarchy is then inserted back into the local database using the *insert document* operator.

[3]Note, "britell:" is automatically added as a prefix by the clone widget to specify the name of the clone and is not part of the resource type.

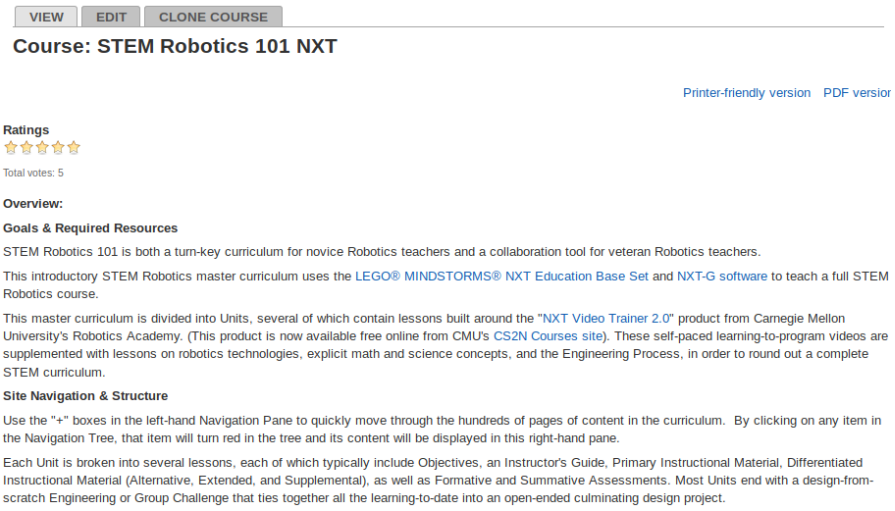


Figure 4.12: The "STEM Robotics 101" course level web page.

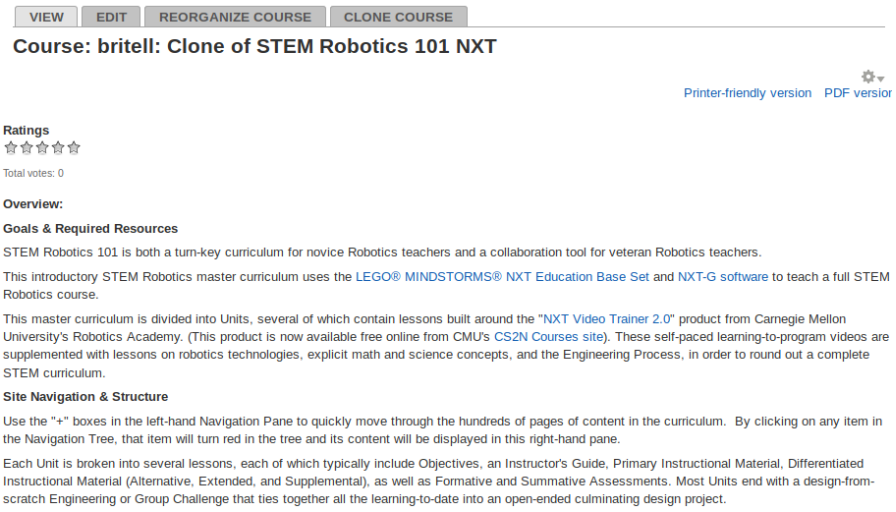


Figure 4.13: A clone of "STEM Robotics 101" course.

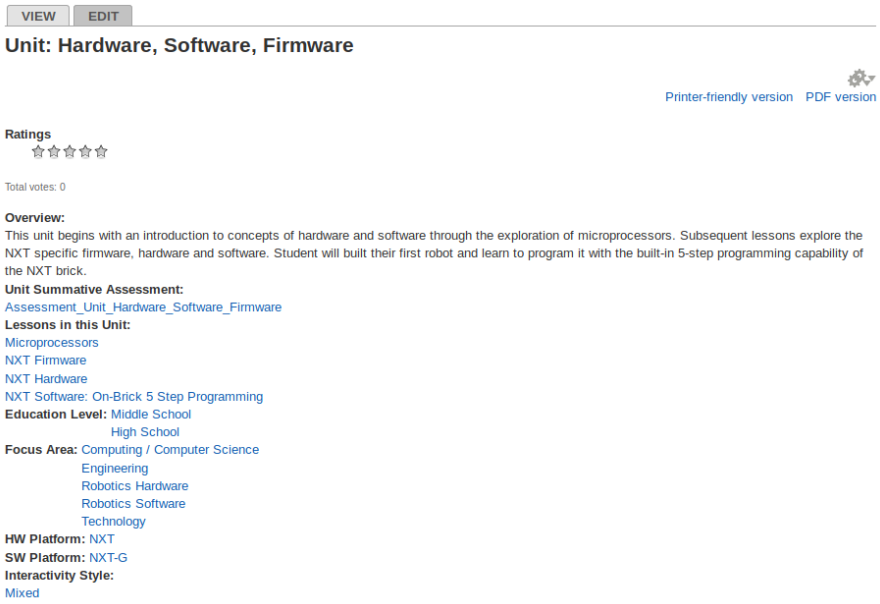


Figure 4.14: The "Hardware, Software, Firmware" unit from "Stem Robotics 101".

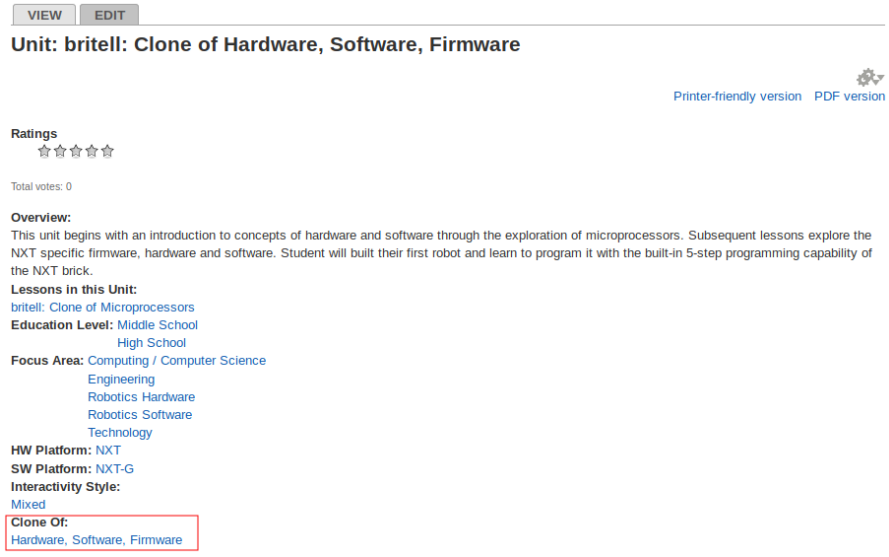


Figure 4.15: A clone of the "Hardware, Software, Firmware" unit.

Lastly, the widget populates the "X-copy-Of" relations, so that the clones can be linked back to their original resources (for example the red boxed link at the bottom of the cloned unit (Figure 4.15) that shows the link back to the original unit). An empty document for the "CloneOf" domain relation is retrieved, populated with the original and cloned ids, and then inserted into the local database.

We can also use the cloning interface (Figure 4.3) as a structural-editing interface to allow users to rearrange their existing content. This function of the widget only requires local documents from the "Has" domain relation, as it only updates the structure of the course and does not need to know the resource attributes. These documents are then updated in the local database to reflect the changes made in the widget.

The Exploration and Comparison Widget

The clone-exploration widget shown in Figure 4.5 uses local documents from the "CloneOf" domain relation and the "Has" domain relation to compare a resource and its clone and also compares one level of hierarchy below these two resources. The widget takes the ids of two resources as inputs. For each input resource, the widget retrieves the children of each resource using the "Has" domain relation. The widget uses the "CloneOf" domain relation to retrieve local documents for the two input resources and all of their children.

The widget then shows if children have been reordered by coloring the lines yellow that connect the root resource to the children. Then, using a basic comparison of the local documents, the widget shows whether any attributes of the clone have been modified from the original resource. If there have been changes, the horizontal line connecting the resources will be colored red, otherwise it will be green.

4.3 RELATED WORK

The combination of the restrictions we impose in our mapping system (i.e., that mappings can only exist between attributes of a single domain relation and a single local relation) and the presence of the full schema and data from local relations in our local documents (even when there are only partial mappings between domain and local relations) allows us to perform inserts and updates from a domain level view while eschewing many of the problems one would typically face due to the view update problem [30] (i.e., can updates against a set of views be translated into correct updates against the schemas over which the views have been defined?). Most relational databases solve the problem by limiting updates over views to cases where the resultant rows of the execution of the view can be identified unambiguously in their base tables, similar to the restrictions of our mappings. Other solutions to the problem, such as Relation Lenses [9] or the channel in the GUAVA [78] system, limit the operations that can be used to create views to sets of bi-directional operators that are known to be updatable. Both of these solutions support updating views over arbitrary joins created with their respective operators. While our mapping restrictions limit a domain relation to act as a view over a single local relation, creating a view at the domain level in our system that arbitrarily joins domain relations will still be updatable as long as the domain attributes have not been aggregated in a way that removes the nested "meta" attribute from the view.

Our use of self-describing local documents is inspired by the many standards and systems used for data exchange and processing—most notably XML [88]. The combination of self-describing documents and relational databases has also been extensively studied and built into most major relational databases, such as the SQL/XML standard introduced in SQL:2003 [34] used in Oracle [64] and PostgreSQL [66] database systems, and pureXML in IBM DB2 [68]. These systems

allow users to store, query, and update XML documents as well as transform relational data into XML formats and vice versa. XML views can also be created that map relational tables to XML data structures that can be used in XPATH [89] queries and used to update relational data. Our local documents are, in essence, a complete view of the local relations to which the domain relation has been mapped. These views suffer from the same view update problems listed above, as well as additional complexity due to the differences between the flat relational model and the hierarchical nature of XML. To solve this problem, these systems typically limit updatable XML views to those where there is an unambiguous mapping between the XML view and the relational database, similar to the restrictions that we impose through our mapping system.

Modern databases and NoSQL document stores [23] often use the JSON [45] self-describing document format to provide access to semi-structured or unstructured data. Much work has gone into providing relational, SQL, access to these types of data stores [3, 22, 82]. This is typically to enable system compatibility and to provide relational-like query interfaces instead of the diverse programmatic query paradigms of each system. We take the opposite approach of providing the self-describing view of our various relational sources for programmatic use.

4.4 CHAPTER SUMMARY

In this chapter we introduced local documents. We have shown how to update and insert local data through domain widgets using populated and empty local documents. We also demonstrated how all data from a mapped local relation (whether or not all the local attributes have been mapped) can be accessed generically using local documents. We formally defined the *local document* operator, *empty document* operator, and *insert document* and *update document* operators.

We presented the use of the new operators in the "STEMRobotics" digital curricular repository, highlighting common use cases for these operators that can

be easily transferred to any domain that requires the reuse and restructuring of existing data.

While this chapter presents the use of the *local document* and *empty document* operators from the domain level, the operators can be used from the canonical level in an identical fashion. Redefining the operators to work at the canonical level requires replacing the *apply* operator in their definitions with the *canonical apply* operator. Updated versions of the operators are shown in Appendix A.

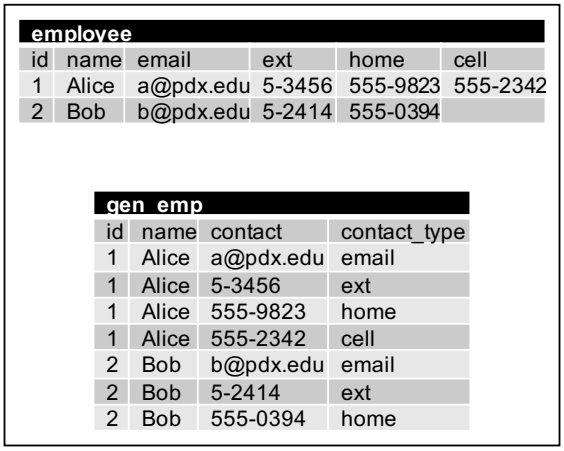
Chapter 5

EXTENDING LOCAL RADIANCE TO SUPPORT DATA-METADATA TRANSFORMATIONS

5.1 INTRODUCTION

One of the key aspects of local radiance is the ability to show local metadata in domain and canonical query results (using the *type* operator). We have shown that non-technical content authors can create mappings and that those mappings can be used in widgets that use the *type* operator (τ) to perform basic metadata to data transformations. In addition, we allow content authors to perform more complex database transformations. In this chapter, we explore using our system to perform both metadata-to-data and data-to-metadata transformations.

The standard DB *unpivot* operation has been studied extensively in the context of databases [28, 86] and information integration, schema integration, and data exchange [40, 47, 87]. The *unpivot* operation moves information from schema (metadata) to data as shown in Figure 5.1, moving from top to bottom. The top of the figure shows the schema in a classical form for an employee table (simplified here), with attributes for id, name, email, ext (extension), home (phone), and cell (phone). The unpivoted version of this table is shown on the bottom of the figure; *email*, *ext*, *home*, and *cell* (formerly attribute names) have been unpivoted and appear in the data. One can choose to unpivot however many attributes one wants except the key; here, for example, the name attribute is not unpivoted. Each employee row on the top has multiple rows on the bottom—one for each of the non-null, unpivoted attributes. Conversely, the standard DB *pivot* operation



employee

id	name	email	ext	home	cell
1	Alice	a@pdx.edu	5-3456	555-9823	555-2342
2	Bob	b@pdx.edu	5-2414	555-0394	

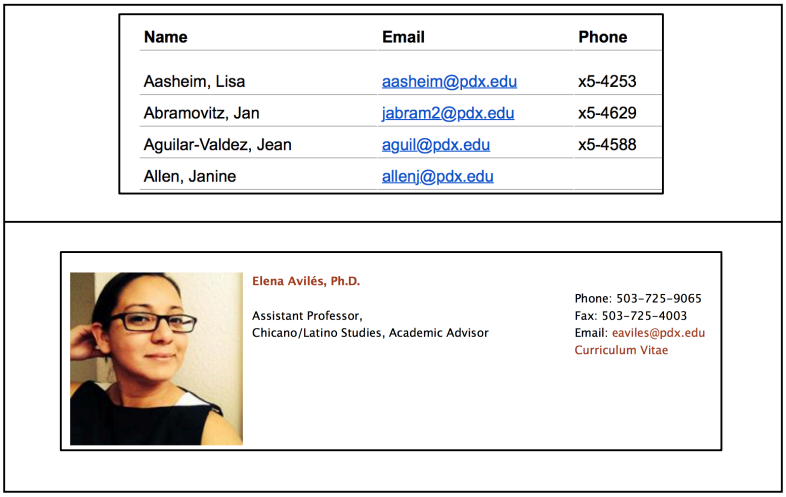
gen_emp

id	name	contact	contact_type
1	Alice	a@pdx.edu	email
1	Alice	5-3456	ext
1	Alice	555-9823	home
1	Alice	555-2342	cell
2	Bob	b@pdx.edu	email
2	Bob	5-2414	ext
2	Bob	555-0394	home

Figure 5.1: Above, a standard schema; below, a schema where the *email*, *ext*, *home*, and *cell* attributes have been unpivoted into a single *contact* attribute and the metadata (i.e., attribute names) from the employee table is transformed into data in the *contact_type* attribute in the *gen_emp* table.

transforms data with a schema similar to the one on the bottom (consisting of id, name, attribute name, attribute value) into data with a schema like the one on the top, moving information from data to schema (metadata).

We believe that structured information shown on a web page presents a conceptual model of the data being displayed. Even for simple, structured data, e.g., contact information on a public web site for employees at a university, the conceptual model can vary, based on the choices made with regard to data versus metadata. That is, the web page might display (possibly a mix of) unpivoted as well as classical forms of data. Consider the widgets from public web pages showing directory information for university personnel in Figure 5.2. The upper widget shows a classical conceptual model for an employee where the schema is shown as column headers. The bottom widget in Figure 5.2 shows a mix of classical and unpivoted data. Notice that the unpivoted attribute names *Phone*, *Fax*, and *E-mail* are shown immediately preceding the data value, rather than in a column header,



Name	Email	Phone
Aasheim, Lisa	aasheim@pdx.edu	x5-4253
Abramovitz, Jan	jabram2@pdx.edu	x5-4629
Aguilar-Valdez, Jean	aguil@pdx.edu	x5-4588
Allen, Janine	allenj@pdx.edu	

Figure 5.2: Above[a], a directory web widget using a classical schema (*Name*, *Email*, *Phone*). Below[b], a directory web widget where the *Name* and *Title* attributes are in a classical format but the *Phone*, *Fax*, and *Email* attributes have been unpivoted.

[a]http://www.pdx.edu/education/gse-faculty-and-staff-directory, accessed 3-17-2016

[b]http://www.pdx.edu/chla/faculty-staff, accessed 3-17-2016

analogous to the DB *unpivot* operation. We also see that the faculty name and title are shown as data values only (without schema information).

The second and fourth columns of Figure 5.3 also have a classical structure, with the schema name in the column header and data shown in the rows. But the first column and third columns in Figure 5.3 each display data of two different types drawn from two different attributes in the underlying local, classical schema: *name* and *rank* are both shown in the column labeled *FACULTY* and *email* and *phone number* are both shown in the column labeled *CONTACT INFORMATION*. In order to transform data from a classical data model (as shown in the top of Figure 5.2) into the form shown in Figure 5.3, a user must combine the two attributes (*name* and *rank* or *email* and *phone number*) into a single attribute (*FACULTY* or *CONTACT INFORMATION*) and then display both the results in a single row for each entity.

FACULTY	OFFICE	CONTACT INFORMATION	SPECIALTY AREA
Raúl Bayoán Cal Associate Professor	EB 402N	rcal@pdx.edu 503-725-2992	Thermal & Fluid Science
Zhiqiang (Tony) Chen, Research Associate Professor, Manager of CEMN	SB1 Room 33	zhiqiang.chen@pdx.edu 503-725-9712	Materials Science
Faryar Etesami Associate Professor	EB 402J	etesamif@pdx.edu 503-725-3261	Design & Manufacturing

Figure 5.3: A university webpage[c] where columns 1 and 3 contain unpivoted data and columns 2 and 4 are normal.

[c]http://www.pdx.edu/mme/faculty-directory, accessed 3-17-2016

These web pages suggest that these conceptual models, with varying amounts of classical and unpivoted data, can be easily understood by end-users.

The focus of this chapter is on allowing domain specialists to fluidly move data of interest in and out of the schema, using data-metadata transformations, including the ability to pivot and unpivot data. In this chapter we make the following contributions:

- We show how correspondences between domain structures[1] and local schemas can support data-metadata transformations.
- We present a case study that shows a complex, faceted browse widget in a digital library that uses data-metadata transformation.
- We extend our simple correspondences to include a predicate, in order to support the classical database *pivot* operation.
- We compare our system against similar systems that perform data-metadata transformations.

[1]As in the previous chapter, at the time of this work we believed we only needed domain structures. This chapter therefore references domain structures as the end query model and not canonical structures.

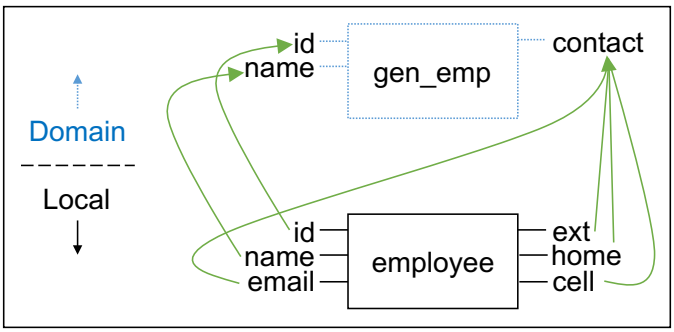


Figure 5.4: A local employee schema (below) is mapped to perform an *unpivot* operation to a generic employee domain structure (top).

5.2 UNPIVOT (METADATA-TO-DATA)

To see how simple correspondences (those without predicates) and domain structures used in our system can support an *unpivot* operation, consider Figure 5.4. The local schema, shown at the bottom of the figure, has a classical structure with five descriptive attributes plus the *id* attribute for the *employee* entity. The domain structure at the top shows a generic employee entity (named *gen_emp*) with an *id* and *name* attribute and an attribute called *contact*. In this example, the local *id* and *name* attributes have been mapped to the *id* and *name* attributes in the domain structure, respectively. The *email*, *ext*, *home*, and *cell* attributes are all mapped to the *contact* attribute in the domain structure. These four correspondences to the *contact* attribute do part of the *unpivot* operation; they combine data from the four local attributes into a single attribute in the domain structure. We can use the *type* operator to perform the rest of the *unpivot* operation.

The queries needed to transform the employee table based on these correspondences are shown in Figure 5.5. The *apply* (α) operator operates on the generic employee entity in the domain structure (*gen_emp*) to produce the intermediate result shown in the middle of the figure. The result of the *type* operator (τ) is then natural joined to this intermediate result (joining on *id* and *contact*) to extract the

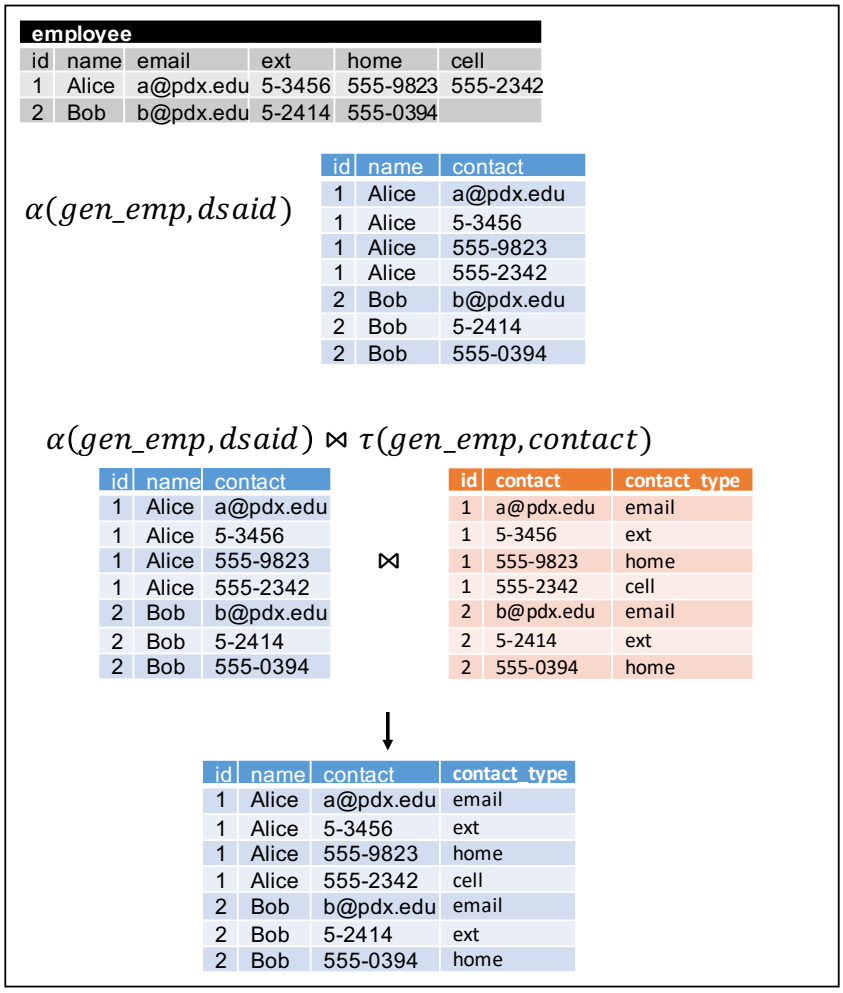


Figure 5.5: An *unpivot* using our query operators and the correspondences and domain structure shown in Figure 5.4

local type (schema) name from the local schema for the data values that appears in the *contact* attribute. The final result is shown at the bottom of the figure.

5.2.1 Case Study: Ensemble and Faceted Browse

As part of the Ensemble[2] project, we helped develop a number of digital library collections in the Ensemble portal. The portal was limited to standard browsing and searching features. The bottom half of Figure 5.6 shows the basic ER model of collections in Ensemble (with a subset of the full attribute set). The portal has two entities (*collection* and *dublin core record*) with the single *contains* relationship.

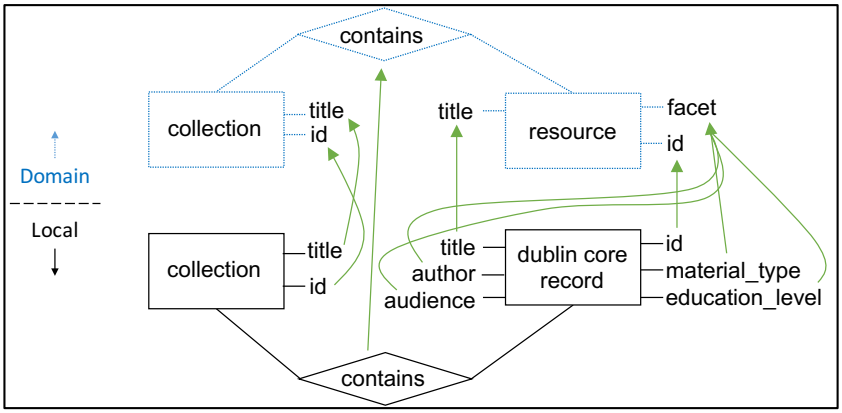


Figure 5.6: The local schema (bottom) for collections in the Ensemble portal and the domain structure (top) used for the faceted browse widget.

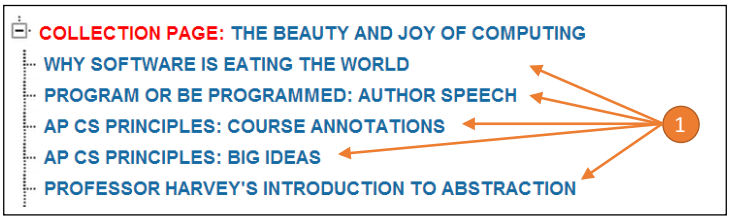


Figure 5.7: An hierarchical navigation widget in the Ensemble portal without faceting.

A collection of digital library records is shown in the standard Ensemble hierarchical navigation widget in Figure 5.7 with the collection entity instance entitled

[2]http://computingportal.org, accessed 11-10-2019

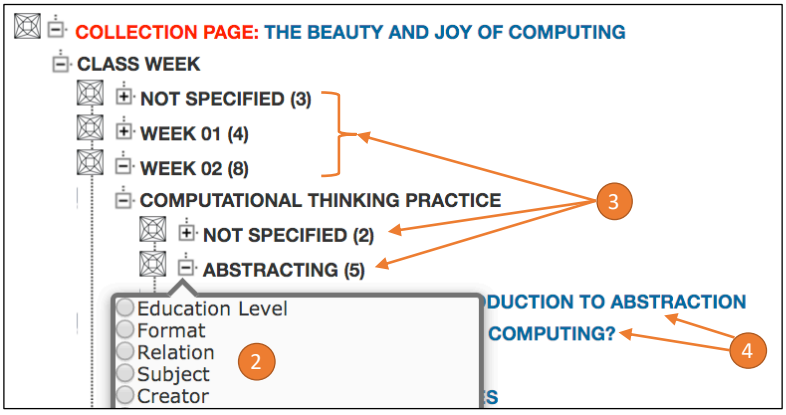


Figure 5.8: A faceted-browse widget in the Ensemble portal where the collection has been faceted by "Class Week" and then "Week 02" has been faceted "Computational Thinking Practice". By clicking the facet diamond next to the plus or minus symbols, a user can further facet the relevant sub-hierarchy. The circled 2 shows the facets available for sorting the resources below the "Abstraction" heading. Each facet shows the count of resources underneath it (the circled 3). Leaf level resources are shown by the circled 4.

"The Beauty and Joy of Computing"[3], a curriculum for introductory computer science, with all of its educational resources. Given the simple ER structure in the local schema, resources could only be browsed as a basic list under a collection (the circled 1).

To facilitate browsing of collections, we leverage the use of *unpivot* in our local radiance system to implement a faceted-browse widget—where the collection in the hierarchical navigation widget can be partitioned at any level by any of the attributes of the resources in the collection. Figure 5.8 shows the same collection after it has been faceted by class week. The new symbol to the left of the plus or minus symbol is our facet symbol. After being faceted by week, we see that we can now also facet any week by any of the remaining attributes that have been mapped to the *facet* domain attribute (as shown in Figure 5.6). For

[3]http://computingportal.org/node/11172, accessed 11-10-2019

example, we see that "WEEK 02" has been faceted by computational-thinking practice. Each level of the hierarchy is able to be faceted differently, enabling users to quickly see resources partitioned by any combination of facets. The "Abstracting" computational-thinking practice could be further faceted by the facets listed in the drop down menu (the circled 2) shown in the figure, e.g., "Education Level" or "Format".

We show how we can use the domain structure and correspondences from Figure 5.6 and our query language in a widget to build our faceted browsing interface.

First, to build the original hierarchical browsing structure (Figure 5.7) we return all resources in the collection (the circled 1) with a collection id of *cid* using the *Resources* function defined below.

$$Resources(cid) = \pi_{resource\_id,resource\_title}(\\ resource \underset{resource.id=contains.resource\_id}{\bowtie} (\sigma_{collection\_id=cid}(\alpha(contains, dsaid))))$$

The *apply* operator on the *contains* domain structure returns all resource_ids (the projection) in the correct collection (the selection). For example, to produce the widget for Figure 5.7, we need to retrieve all the resources in "The Beauty and Joy of Computing" collection with id "11172" (this id is used by the system internally). The results of the *Resources* function is shown below.

Result Set 5.1

```
# resources(11172)
  id   |    title
-------+---------------------------
 11172 | Why Software is Eating the World
 11173 | Program or be Programmed: Author Speech
 11174 | AP CS Pinciples: Course Annotations
 11175 | AP CS Principles: Big Ideas
 11176 | Professor Harvey's Introduction to Abstraction
```

The widget then uses this result to populate the navigation tree.

Next, we find all facet types and values used in the collection with id *id* by joining the *Contains* domain relationship with the *Resource* domain entity on *resource_id* for a collection with id *cid* using the *Facets* function defined below. This function returns all the facet types and values in the given collection using

the τ operator.

$$Facets(cid) = \pi_{facet\_type}(\sigma_{id=cid}(\tau(resource, facet, dsaid)))$$

For example, the facets for "The Beauty and Joy" collection are as follows:

Result Set 5.2

```
# facets(11172)
facet_type
------------
Class Week
Computational Thinking Practice
Education Level
Format
Relation
Subject
Creator
```

The widget then uses this result to rebuild the navigation tree. The widget also stores which facets have already been used in a given tree path and makes sure they are not duplicated. For example, Figure 5.8 shows the tree has already been faceted by "Class Week" and "Computational Thinking Practice", so those facets do not appear in the facet list produced at the circled 2.

Once we have all of the facet types and values (i.e., an unpivot) the widget creates the faceted-browse interface by providing the count of the resources within each facet (as shown in the circled 3 in Figure 5.8). To do so, we first define the *Facet_Resources* function that given a facet type (*ft*) and a value (*fv*) we can find all resources in a given collection (*cid*) that have that facet value as follows:

$$Facet\_Resources(cid, ft, fv) = \pi_{resource.id}\Bigg(\sigma_{\substack{facet\_type=ft \wedge facet=fv \wedge \\ resource.id \in \pi_{resource\_id}(Resources(cid))}} \big(\alpha(resource, dsaid) \bowtie \tau(resource, facet, dsaid)\big)\Bigg)$$

The count of resources below a facet value in the tree is then determined by the path of facet types and values from the root of the tree, which we call the *facet path*. The count of resources with a given facet value existing in a facet path consisting of the facet types and values $((ft_1, fv_1), \ldots (ft_n, fv_n))$ is found with the following query that uses the standard relational algebra extended with the grouping operator [39] (γ). If the facet path exists then the query is modified by

filtering the resource ids based on the conjunction of facet types and values in the tree path by modifying the select clause of the query.

$$\gamma_{facet,Count(resource.id)}\Bigg(\sigma_{\substack{facet\_type=ftype\wedge \\ resource.id\in\pi_{resource\_id}(Resources(cid))\wedge \\ resource.id\in Facet\_Resources(id,ft_1,fv_1)\wedge \\ \dots \\ resource.id\in Facet\_Resources(id,ft_n,fv_n)}} \Big(\alpha(resource, dsaid) \bowtie \tau(resource, facet, dsaid)\Big)\Bigg)$$

The widget can then populate the resources under a given facet path by filtering the results of the *Resources* function above, using the same conjunction of facet types and values, producing the results in the circled 4 in Figure 5.8 with the following query:

$$\pi_{resource\_id,resource\_title}\Bigg(\sigma_{\substack{resource.id\in Facet\_Resources(id,ft_1,fv_1)\wedge \\ \dots \\ resource.id\in Facet\_Resources(id,ft_n,fv_n)}} \Big(\alpha(resource, dsaid)\Big)\Bigg)$$

5.3 PIVOT (DATA-TO-METADATA)

In this section we show how our system can be used for the standard *pivot* operation and more generally for data-to-metadata transformation. Figure 5.9 shows a mapping that contains standard correspondences for local *id* and *name* attributes but also pivots local *contact* data into the *ext* domain attribute. In the local schema, the *contact* attribute stores all of the contact information and the corresponding type is in the *contact_type* attribute. We would like this data to appear in a pivoted form, where contact information is broken out into the *email*, *ext*, *home*, and *cell* domain attributes.

In order for to perform this *pivot*, we must tell the system which data from the local schema should end up in the *ext* domain attribute, for example. First recall

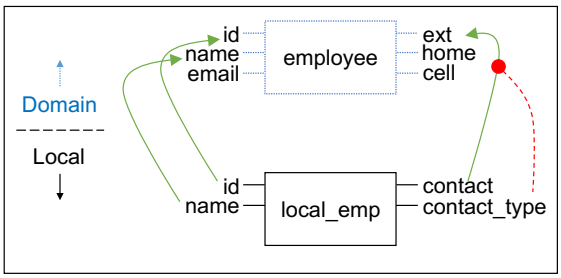


Figure 5.9: An example mapping showing standard correspondences for *id* and *name* attributes and using a conditional correspondence to map local *contact* data into the domain *ext* attribute where the local *contact_type* attribute is equal to "ext".

from Chapter 2, a correspondence in our system is in the domain structure-local DB mappings relation

$ds\_ldb\_m(\underline{id}, ldbid, dsid, dr\_lr\_ms(id, lr, dr, p, corrs(id, la, da)))$

and is of the form

$$corr = (id, la, da)$$

where each correspondence has an id, a local attribute *la*, and a corresponding domain attribute *da*. While a mapping contains the predicate p at the mapping level, in order to perform a *pivot* operation, we need to specify a predicate at the correspondence level.

A conditional correspondence adds a predicate *cp* to the correspondence and has the form

$$c = (id, la, da, cp)$$

and we extend the definition of the local DB to domain structure mappings relation to $ds\_ldb\_m(\underline{id}, ldbid, dsid, dr\_lr\_ms(id, lr, dr, p, corrs(id, la, da, cp)))$

Then, when the correspondence is used in an *apply* operation, data from the local attribute *la* will only be in the query result for domain attribute *da* when the predicate *cp* evaluates to true. We make one small change to the *apply* operator; where the *select* operator in line 2 of the **proj_type_nest** function (Function 3.1.4 in Chapter 3) was previously "$\sigma_{dr\_lr\_m.p}$" it is now "$\sigma_{dr\_lr\_m.p \wedge corr.cp}$".

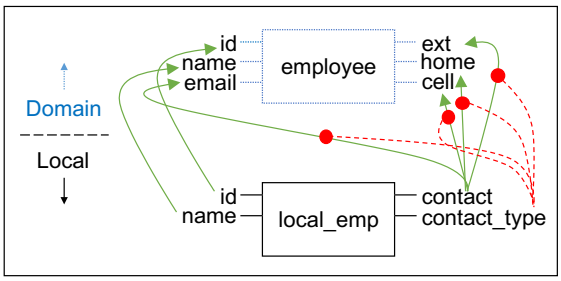


Figure 5.10: The complete set of correspondences to pivot the local schema into the domain structure. A user can create a regular correspondence and then chose to add a condition (in this case the specific pivot conditions) for the contact attribute correspondences.

We add new visual syntax for (a limited form of) the correspondence level predicate, shown in Figure 5.9; a regular correspondence (the solid line) is augmented by the dot with a dotted line. This visual syntax is translated into a predicate for the correspondence where data from the local attribute with the solid line will appear in the domain attribute only when data in the local attribute with the dotted line is equal to the name of the domain attribute. The correspondence shown in Figure 5.9 results in the predicate

$$cp = (contact\_type = \text{“}ext\text{”})$$

In Figure 5.10, we show the complete set of correspondences to pivot from the local schema to the domain structure. The end-user can easily combine regular correspondences and conditional (dotted) correspondences in a single mapping. In this case the *id* and *name* attributes are mapped directly (without correspondence predicates) while the *email*, *ext*, *home*, and *cell* attributes are pivoted from the local *contact* attribute.

Figure 5.11 shows an *apply* operation on the domain structure that uses the correspondences, takes data from multiple tuples in the source database, and returns a single tuple in the query answer based on the correspondences. For example, in the figure we see that four tuples for “Alice” in the *local_emp* table are combined

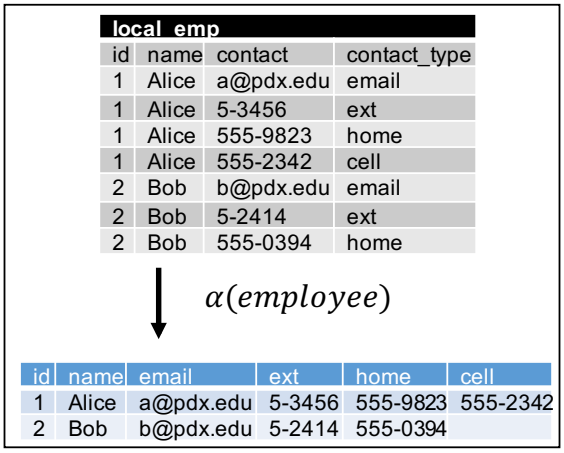


Figure 5.11: The *pivot* operation, using the local and domain structures from Figure 5.10 with example employee data.

to make one tuple in the output query; these tuples are joined based on the *id* attribute.

5.4 COMPARISON TO OTHER SYSTEMS AND RELATED WORK

Table 5.1 shows a comparison of our our system (LR) to SchemaSQL [47], FIRA/FISQL[86, 87], Clio [40], GUAVA [78], and the *unpivot* and *pivot* operations supported in SQL (in systems such as Oracle [63] and SQL Server [56]).

While all these systems can do *pivot* and *unpivot* operations, we see that SQL is quite limited and the syntax is complex. The other three systems allow more generalized transformations and Clio (as well as our system) provides a simple visual syntax. SQL, SchemaSQL, GUAVA, and our system produce a single non-ambiguous result, whereas FIRA and Clio can potentially have ambiguous or non-intended results; FIRA relies on the optimal tuple merge (which may not be unique) and Clio generates many different mappings that may or may not be correct. Our system avoids the ambiguity problem by the restrictions imposed by our mapping system, limiting users to simple correspondences and maintaining ids. We have

Table 5.1: Comparison of Data-Metadata Transformation Systems

	SQL	Schema-SQL	FIRA/ FISQL	CLIO	GUAVA	LR
Can perform Pivot and Unpivot	✓	✓	✓	✓	✓	✓
Can perform arbitrary metadata-data transformations	✗	✓	✓	✓	✓	✓
Can perform arbitrary data-metadata transformations	✗	✓	✓	✓	✓	✓
Has a simple visual syntax	✗	✗	✗	✓	✗	✓
Has a non-ambiguous result	✓	✓	✗	✗	✓	✓
Has preview capability	✗	✗	✗	✗	✓	✓

also explicitly built in the preview mechanism for our system; while this could be implemented on top of the other systems (beyond GUAVA which already has it), it is not by default.

One of our main goals is to bring these operations to non-technical users by making it accessible through a mapping interface. We take much inspiration from Clio in this regard as opposed to the rest of these systems which target database administrators and developers who must have deep knowledge of SQL and these systems or, in the case of GUAVA, the channel mechanism.

5.5 CHAPTER SUMMARY

In this chapter, we have shown how local radiance can be extended (specifically by extending the definition of correspondences to include a predicate) to encompass standard *unpivot* and *pivot* operations. We have shown how the *unpivot* operation can be achieved through the combination of mappings and the *type* operator. The faceted-browsing widget demonstrated a real world use case of the *unpivot* operation with our mapping and query interface to create the dynamically facetable navigation tree. We defined conditional correspondences and presented how they can be used to perform the *pivot* operation.

Chapter 6

IMPLEMENTATIONS

In this chapter we discuss the different implementations of local radiance systems. We show how local radiance has been used across a number of platforms in a number of scenarios. We show how the system has evolved and discuss lessons learned along the way.

Table 6.1 lists the different implementations of local radiance with the reason for creating each implementation, how each version was implemented, and limitations and lessons learned from each version. We discuss each of these versions in detail in this chapter.

Table 6.1: Implementations

Version	**Strengths**	**How**	**Limitations**
Initial Drupal. 2500 lines of code. In use since 2011.	Able to represent multiple local schemas with local radiance in generic widgets	Hard-coded canonical structures and mappings in widget code. Queries plus widget code used to perform local radiance.	Not easily extendable or maintainable. Navigation widget had to be in place for other widgets to use mappings and canonical structures.
WordPress. 600 lines of code. Prototype never used in production.	Shows that local radiance in generic widgets is feasible in multiple web CMSs	Direct port of initial Drupal implementation.	Same problems as the initial Drupal implementation.

Table 6.1 – *Continued from previous page*

Version	**Strengths**	**How**	**Limitations**
Drupal Query Interface. 1700 lines of code. In use since 2012.	Demonstration of first HERM-based formalism. Allows for error checking formalism. Creates reusable domain structures. Local radiance is defined in queries, not mixed with widget code.	Domain structures are stored as files so they can be reused across widgets and applications. Extended relational query interface with *type* and *apply* operators.	Mappings need to be written by developers instead of end-users. Query interface requires knowledge of building relational algebra query trees.
Drupal Mapping Interface. 700 lines of code. In use since 2014.	Provides an end-user mapping interface to bring local radiance to domain users, used in our user test.	Graphical interface built to allow web users to add mappings. Provides preview of widgets to error check mappings. Mappings are stored in database.	Still requires some knowledge of the Drupal site structure beyond simply knowing domain schema.
Widget Specifications. 500 lines of code. Not in public use, in prototypes since 2014.	Allows users to customize widgets while mapping.	Creates default parameters for widget code that can then be overwritten by end-users while performing mappings. Widget specifications stored in database.	Requires understanding of some underlying Drupal code.
PostgreSQL. 700 lines of code. Developed in 2018.	Implementation of nested relational model-based formalism for error-checking and refinement. Local radiance queries written in SQL.	Formal definitions of local schema, domain structures, canonical structures, and mappings are stored as relations. *Type*, *apply*, *canonical apply*, *local document*, *empty document*, and *apparent* operators implemented as PL/pgSQL functions. All operators can be used with standard SQL queries.	Limited to a single database.

6.1 FIRST DRUPAL ITERATION

The local radiance system was first developed to solve the problem of how to generically present information in the STEMRobotics[1] digital repository that we developed, comprised of middle and high school robotics curricula. STEMRobotics is a publicly accessible digital library. We host 6000+ resources created by some of our 4300+ registered teacher-users. We accept (and encourage) content of any form or structure that the teachers wish to create. Responding to the diverse needs of our user base has been a motivating factor of much of the work in this thesis. Figure 6.1 shows the navigation tree for four different courses in the site. For each course in the site, regardless of type, the widget shows the local course type (e.g., "Course", "Tutorial Course", "Standard", or "Chalenge-based") and the local types (e.g., "Guide", "Unit", "Challenge", etc.) for each part of the course. In Figure 6.1, the widget instance in the top left shows a course that contains units with lessons whereas the widget instance in the top right shows a course with units where the units contain instructional materials. The local type information can be useful when navigating curricular materials since, as an example, understanding whether an assessment is used as a challenge, as an assessment, or as instructional material may help a teacher decide how to use it in their classes. For example, in Figure 6.1 an assessment may be used as a summative assessment for a unit in the course in the top left, an assessment resource in the class in the lower right, or as a challenge in the course in the lower left.

Being built upon the Drupal content management system both facilitated development and presented challenges. Drupal is built upon an underlying relational database. Every content type in Drupal is a subclass of the base "Node" class, which means that every content type has some standard attributes such as a url, a title, a type, an author, and a creation date. Any extension to the base "Node"

[1] http://stemrobotics.cs.pdx.edu

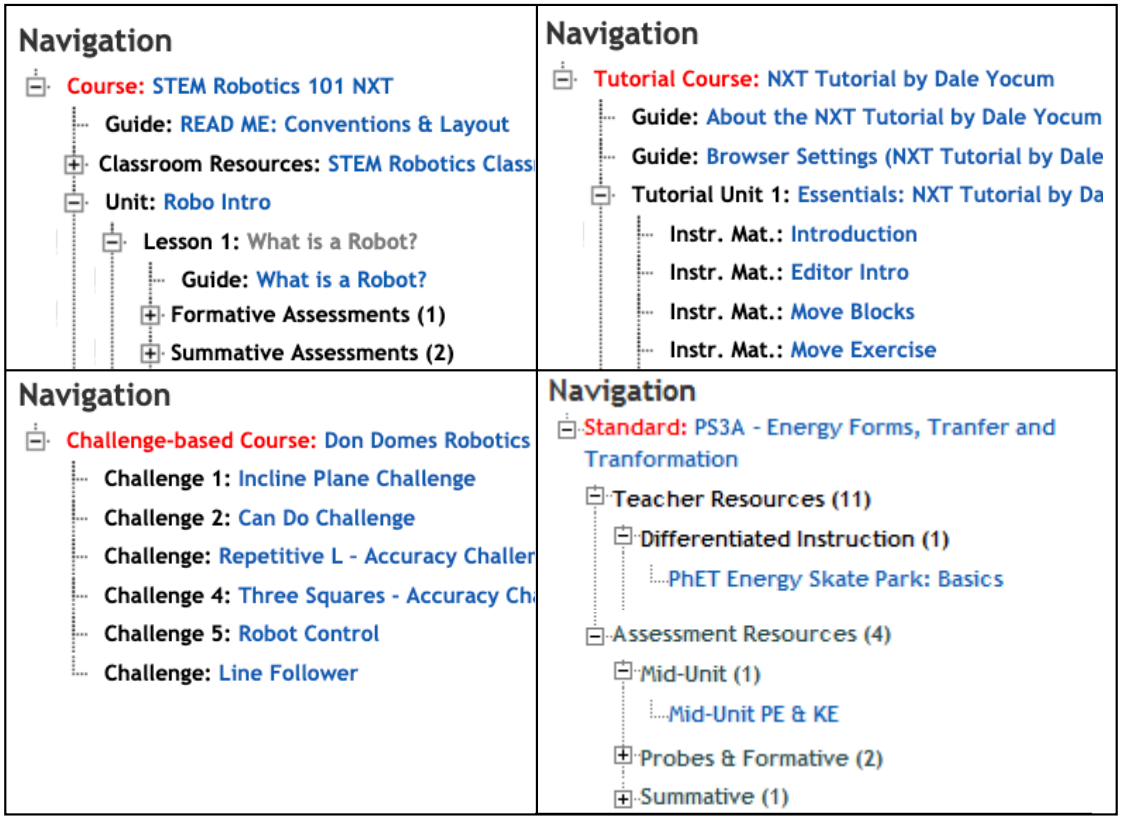


Figure 6.1: The navigation widget in STEMRobotics generically shows different course types with local type information.

type is then separated out into its own relation, which includes any attribute not in the base class as well as any relationship between one content type and another. For example, Figure 6.2 shows the "field_data_field_overview" relation in the Drupal database that stores data for the "overview" attribute for the "lesson" content type (this type is shown in the "bundle" field). The "entity_id" field stores the node id for the lesson associated with the given overview attribute. Figure 6.3 shows the "summative assessment" relationship between the "lesson" content type and the "assessment" content type. Here there exists a relationship between the lesson with id "198" and two assessments "205" and "206". The base "Node" relation is useful in creating generic widgets, since we can always find the base attributes and content types without mappings. As shown Figure 6.1, the relationship type

```
mysql> select * from field_data_field_overview;
+-------------+--------+---------+-----------+-------+-----------------------------------------------------+
| entity_type | bundle | deleted | entity_id | delta | field_overview_value
+-------------+--------+---------+-----------+-------+-----------------------------------------------------+
| node        | lesson |       0 |       198 |     0 | The goal of this lesson is to draw out student's
                                                        preconceptions of robots and explore the variety
                                                        and ambiguity of "What is a Robot?"
| node        | lesson |       0 |       207 |     0 | The goal of this lesson is to transition from
                                                        discussing robots in general to the specifics of
                                                        NXT robot.
```

Figure 6.2: The overview attribute for the "lesson" content type is stored in the "field_data_field_overview" relation in the Drupal backend database.

```
mysql> select * from field_data_field_summative;
+-------------+--------+---------+-----------+-------------+----------+-------+---------------------+
| entity_type | bundle | deleted | entity_id | revision_id | language | delta | field_summative_nid |
+-------------+--------+---------+-----------+-------------+----------+-------+---------------------+
| node        | lesson |       0 |       198 |         203 | und      |     0 |                 205 |
| node        | lesson |       0 |       198 |         203 | und      |     1 |                 206 |
```

Figure 6.3: The summative assessment relationship between the "lesson" content type and the "assessment" content type is stored in the "field_data_field_summative" relation in the Drupal backend database.

is also useful, e.g., to show how an assessment is used.

The first implementation added a relation in the underlying database for each canonical structure and then hard-coded a large number of queries (as shown in Figure 6.4) to populate these relations. The top query adds the node id of all course types ("curriculum", "tutorial_course", ...) to the "course" canonical structure ("course_cs"). Then the "structural unit" ("su_cs") canonical structure is populated for each course type with the first level of hierarchy in each of the courses. These relations were then used to build generic widgets, such as the navigation widget above, the aggregation widget (the left side of Figure 6.5, used to show our users metadata information about resources and their descendants), and a search-in-context widget (the right side of Figure 6.5, which helps users understand the context of resources retrieved through a search result).

This implementation was built primarily around the need for a generic navigation widget. As such, all the queries to create canonical structures and their

```
$result = db_query("INSERT INTO course_cs
                    SELECT nid
                    FROM node
                    WHERE type = 'curriculum' OR type = 'tutorial_course' OR type = 'challenge_based_course' OR type = 'concept_based_course';");
//drupal_set_message(t('courses inserted into course_cs successfully'));
$result = db_query("INSERT INTO su_cs
                    SELECT field_element_nid as 'unit', field_instr_guide_nid as 'ig', nid as 'course'
                    FROM node n
                    JOIN field_data_field_element u
                      ON (n.nid = u.entity_id)
                      LEFT OUTER JOIN field_data_field_instr_guide ig
                      ON (u.field_element_nid = ig.entity_id)
                    WHERE n.type = 'curriculum';");
$result = db_query("INSERT INTO su_cs
                    SELECT field_tut_units_nid as 'unit', null, nid as 'course'
                    FROM node n
                    JOIN field_data_field_tut_units u
                      ON (n.nid = u.entity_id)
                    WHERE n.type = 'tutorial_course';");
$result = db_query("INSERT INTO su_cs
                    SELECT field_concept_unit_nid as 'unit', null, nid as 'course'
                    FROM node n
                    JOIN field_data_field_concept_unit u
                      ON (n.nid = u.entity_id)
                    WHERE n.type = 'concept_based_course';");
$result = db_query("INSERT INTO su_cs
                    SELECT field_challenges_nid as 'unit', null, nid as 'course'
                    FROM node n
                    JOIN field_data_field_challenges u
                      ON (n.nid = u.entity_id)
                    WHERE n.type = 'challenge_based_course';");
//drupal_set_message(t('units inserted into su_cs successfully'));
```

Figure 6.4: A small subset of the mappings between Drupal content types and the "structural unit" canonical structure (su_cs).

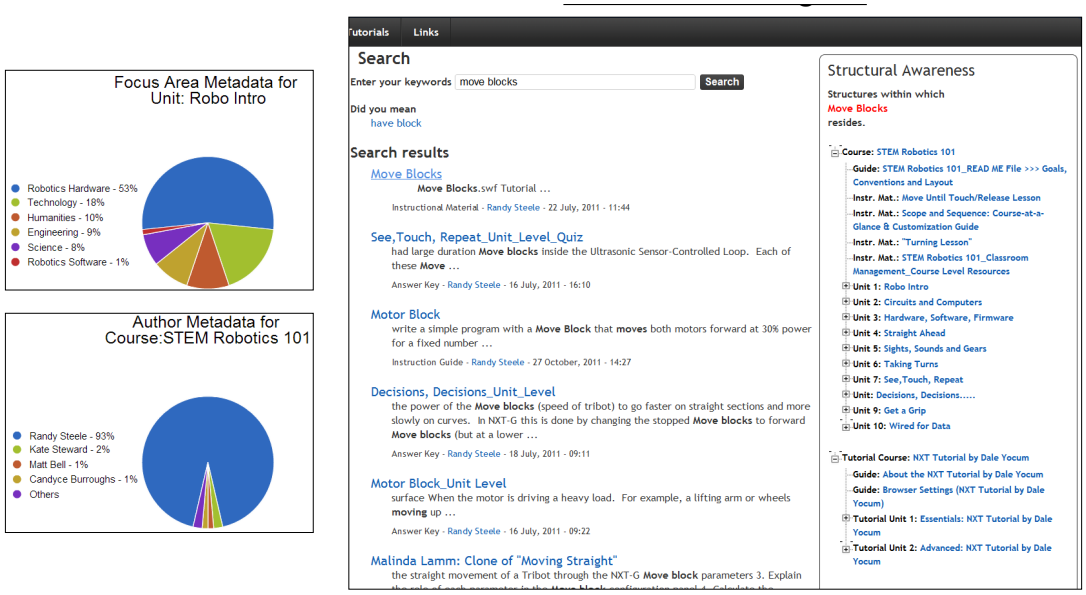


Figure 6.5: Left, metadata information is aggregated and presented for a course and unit in STEMRobotics. Right, when a search result is clicked (under the search results) the "Structural Awareness" tab on the right is populated with all the courses in the site that contain the selected resources (in this case, "STEMRobotics 101" and "NXT Tutorial by Dale Yocum" both contain the "Move Blocks" resource).

mappings were all written to that goal. While the number of courses in the repository was small, this setup worked relatively well, but as the number of types in the repository grew and more students helped develop the system, it became harder to modify and debug. Also, as local radiance features were mixed between code and queries, adding new widgets necessitated duplicating much of the local radiance system with small changes for each specific widget. The original navigation menu created using this implementation has been running in the STEMRobotics repository since 2011 and is still being used today.

6.2 BRINGING LOCAL RADIANCE TO WORDPRESS

Shortly after the first Drupal iteration we also implemented local radiance in WordPress[2] to explore the feasibility of using local radiance beyond Drupal. At the time of development Drupal and WordPress were the two most commonly using web content management systems. Using the Drupal implementation as a reference, we recreated a small subset of the STEMRobotics repository and the navigation widget. WordPress was also built upon a relational database and allowed user-created content types, but in a simpler form than Drupal. Where Drupal separated out all custom attributes and relationships, in WordPress all base attributes are stored in the "wp_posts" relation and all custom-defined type information is unpivoted and stored in the "wp_postmeta" relation. Mappings to populate the canonical structures are then defined as queries as shown in Figure 6.6. All the mappings have the same form, the only change is the "meta_key" value for the custom information.

While the WordPress implementation showed that we could easily port local radiance to other content management systems, it did not address any of the problems from the original implementation and therefore suffered from the same limitations.

[2]http://wordpress.com, accessed 11-10-2019

```
$results = $wpdb->get_results("select post_id, meta_value, post_title
                                from wp_postmeta join wp_posts on
                                (wp_postmeta.post_id = wp_posts.ID)
                                WHERE meta_key = 'belongs'");
foreach($results as $row) {
        $type = get_post_type_object(
                get_post($row->post_id)->post_type)->labels->singular_name;
        $parent = $row->meta_value;
        $title = $row->post_title;
        $part = $row->post_id;
        // Update the database
        $wpdb->insert(
                'qd_navmenu_instance_data',
                array(
                        'parent' => $parent,
                        'part' => $part,
                        'title' => $title,
                        'type' => $type,
                )
        );
}
$courses = json_encode(qd_treemap_buildcourses(null));
```

Figure 6.6: A query to build the parent-part canonical structure in WordPress.

6.3 THE QUERY INTERFACE

In order to address the limitations of the first Drupal implementation, the local radiance system was redeveloped built upon the formalism that was presented at the International Conference on Information Modeling and Knowledge Bases [13]. This formalism was based on the higher-order entity-relationship model (HERM) [79]. We chose HERM as it allowed us to model many aspects of a web CMS that the traditional relational model could not, such as complex attributes (nested attributes, sets, and lists) as well as handling higher-order and cluster-type relationships between one entity type and multiple other entity types. We defined our *apply* and *type* operators using the HERM query algebra. Instead of directly representing local type information in the results of an apply, the HERM-based system only retrieved local information when the *type* operator was used.

The structure of this implementation is shown in Figure 6.7, where domain structures are stored separate (in the "quickdraw_ds" directory) from both the

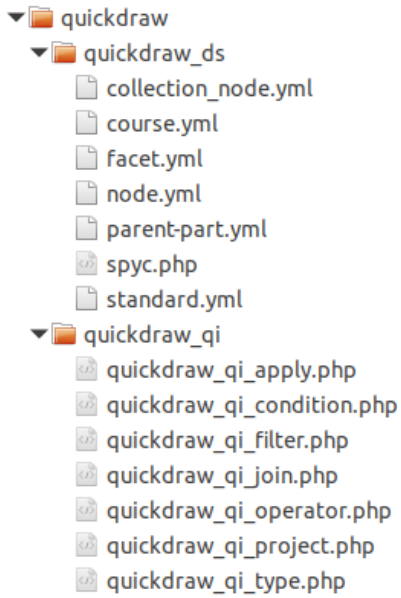


Figure 6.7: Installation structure of the query interface implementation. The query interface is built upon the operators shown in the "quickdraw_qi" directory. Each operator is defined as a subclass of the base "quickdraw_qi_operator PHP class. Domain structures are stored in YAML files in the "quickdraw_ds" directory which can be reused in different applications and instantiations. Examples of the YAML files are shown in Figure 6.8.

widget code and the application database such that they can be used in multiple widgets and in multiple applications. The query interface is built using the operators in the "quickdraw_qi" directory. Domain structures are stored as YAML [90] files (a self-describing data serialization format) that allows defined domain structures to be easily shared between applications and implementations. Examples of the "Parent-Part", "course", and "facet node" domain structures in YAML are shown in Figure 6.8.

The query interface allowed developers to build query trees similar to those used in the GUAVA [78] development system that included *filter*, *project*, and *join* operators, as well as our *apply* and *type* (built into the *apply*) operators. Using the query interface allowed us to divorce the queries used to populate widgets from

```yaml
---
name: Parent-Part
id: 2
type: domain relationship
domain_entities:
  - id: 1
    label: parent

  - id: 1
    label: part
domain_attributes:
  - order
---
```

```yaml
---
name: course
id : 5
type: domain entity
domain_attributes:
  - title
---
```

```yaml
---
name: facet node
id : 3
type: domain entity
domain_attributes:
  - facet
---
```

Figure 6.8: Domain structures (in YAML) are shown for the "Parent-Part" domain relationship (left), the "course" domain entity (center), and the "facet node" domain entity (right).

the rest of widget code. Figure 6.9 shows a query to find all the facets and facet types associated with a course with node id "291". The query first uses the *apply* operator on the "course" domain structure (using id "5" from Figure 6.8). It then uses the *filter* operator to only return course "291". Then a second *apply* operator is created for the "facet node" domain structure (using id "3" from Figure 6.8). The second apply and the filter are then joined and the title, facet, and facet type attributes are projected. Thus, a query tree is built with the root being the *project* operator followed by the *join* operator with the two *apply* operator at the leaf level. A SQL query is then created by running the "makeSQLText" method of the root operator (here, `$project->makeSQLText()`), using a recursive visitor pattern that builds the correct portion of the SQL query for each of the operators.

The query interface was used to create the clone and exploration widgets presented in Chapter 4 as well as the faceted navigation widget presented in Chapter 5. This interface has also been used to creates widgets in the digital library domain [10, 16, 17, 19, 20, 21]. The clone widget has been running in STEM-Robotics since 2013. Widgets based on this implementation ran in the Ensemble[3]

[3]http://computingportal.org, accessed 11-10-2019

```
$course = new qdqiApply(5);
$filter = new qdqiFilter($course,'nid=291');
$facet_node = new qdqiApply(3);
$join = new qdqiJoin($filter,$facet_node);
$project = new qdqiProject($join,{'title','facet','facet_type'});
$result = db_query($project->makeSQLText());
foreach($result as $row) {
        ...
}
```

Figure 6.9: The query to find all facets and facet types for the course with node id "291". The query is built by combining query operators and using the *apply* operator with the domain structure ids from Figure 6.8.

digital library from 2012 to 2015. The CorePlus[4] digital library for secondary technical education from the Boeing corporation was built upon this implementation in 2015 and is also currently running. This implementation has been publicly available[5] since 2013.

While this implementation facilitated the use of domain structures and the local radiance query interface across multiple widgets and applications, mappings still needed to be manually entered into the database using SQL directly or as queries in the interface code. Also, while many developers understand and can use SQL in their code, most do not have the same fluency in relational algebra, so creating relational algebra-like query trees remains as a challenge to some. Also, the use of HERM allowed us to represent the complex structures within the CMS but it then had to be implemented on top of the Drupal backend, which is a relational database. Complex structures (like lists, sets, and nested attributes) needed to be transformed into a relational form.

[4]http://coreplus.cs.pdx.edu, accessed 11-10-2019
[5]https://www.drupal.org/sandbox/britell/2150221, accessed 11-10-2019

6.4 THE MAPPING INTERFACE

To achieve our goal of letting end-users create mappings, we developed the mapping interface shown below in Figures 6.10, and 6.11. This mapping interface was used in the user study presented in Chapter 2.

The mapping interface was built as an extension to the query-interface module. It uses the domain structure specifications shown in Figures 6.7 and 6.8. The interface allows users to choose which domain structure to use while they create mappings in the interface (Figure 6.10). A mapping is then created using the interface shown in Figure 6.11. This interface is built by querying the Drupal system catalog to find all content types and their attributes. When a mapping is created, it is stored in the Drupal database using the HERM-based mapping definition [13] with a mapping id and a set of correspondences. The result set below shows a subset of mappings created in the CorePlus repository. The mapping ids are shown under *mid* and correspondence ids are shown under *cid*. The attribute *dsid* refers to the ids shown in the domain structures in Figure 6.8. The *reltype* and *rel* attributes store information about the Drupal local model. The *label* and *delta* attributes are used for display and ordering of mappings in the interface.

Result Set 6.1

```
mysql> select * from main_quickdraw_base_mappings;
+---+---+----+--------------+---------+-------------------------------+--------+-------+
|mid|cid|dsid| type         | reltype | rel                           | label  | delta |
+---+---+----+--------------+---------+-------------------------------+--------+-------+
| 1 | 1 |  1 | node:course  | field   | field_semester                | c->s   |    -3 |
| 1 | 2 |  5 | node:course  | field   | field_title                   | c->t   |    -3 |
| 3 | 3 |  1 | node:unit    | field   | field_lessons                 | u->l   |    -3 |
| 2 | 4 |  1 | node:semester| field   | field_unit                    | s->u   |    -2 |
| 2 | 5 |  5 | node:semester| field   | field_title                   | s->t   |    -1 |
| 3 | 6 |  5 | node:unit    | field   | field_title                   | u->t   |     0 |
| 4 | 7 |  1 | node:lesson  | field   | field_slides                  | l->s   |    -3 |
| 4 | 8 |  5 | node:lesson  | field   | field_title                   | l->t   |    -2 |
```

The mapping interface can also load widgets that have been enabled in the site, to let users preview their mappings using the widget previewer shown in Figure 6.12. After selecting a mapping and pressing the "Preview Selected Mapping" button, the user is presented with a choice of widgets within which to preview the mapping. Figure 6.12 shows a mapping previewed in the navigation-tree widget.

Figure 6.10: The first screen in the mapping interface allows the users to choose which domain structure that they would like to create a mapping for.

This widget requires that a node id is provided for the root of the tree. The widget will pre-populate the node id with an id from the system that is found through the selected mapping, but the user may also provide their own node id. The navigation widget is then shown.

In Chapter 2 our user study showed that end-users can understand this interface and use it to create mappings. But this interface still requires users to understand some underlying Drupal concepts, such as the way content types are stored and accessed.

The mapping interface was added to the publicly available Drupal module in 2014 and is in use in the CorePlus repository.

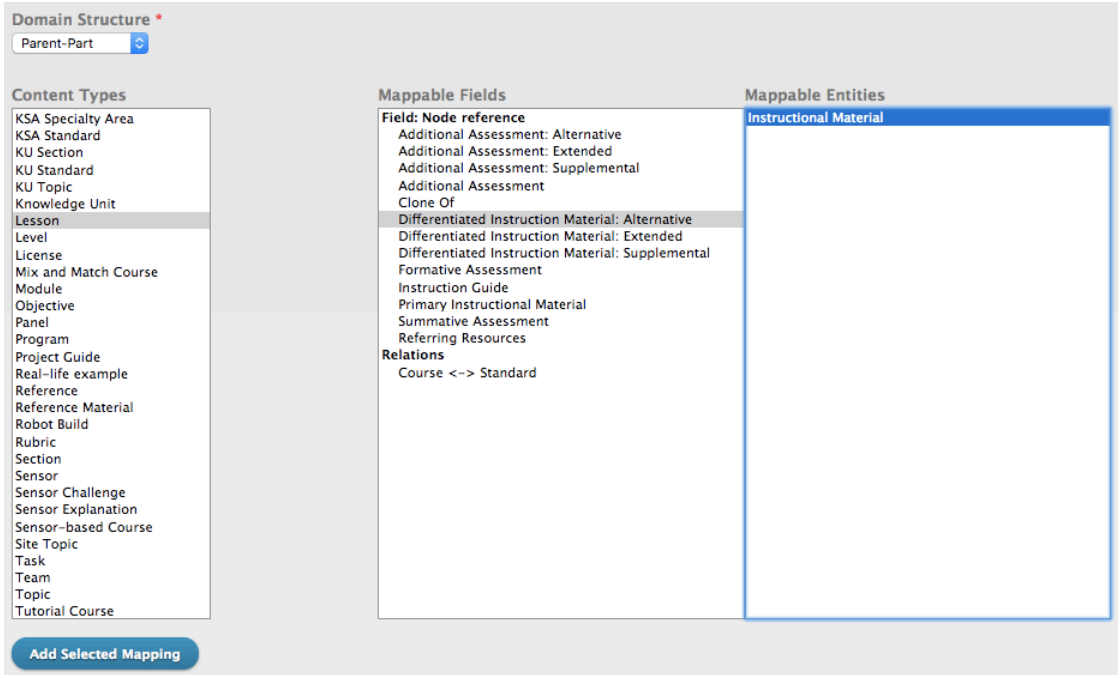


Figure 6.11: Once the user selects a domain structure they are presented with all of the possible content types in the system (left screen). After choosing a content type the user is present with all possible fields for the type (both attributes of the type and relationships to other types; middle screen). If the chosen field is a relationship the user is presented with a choice of related content type (since Drupal allows higher order relationship types; right screen).

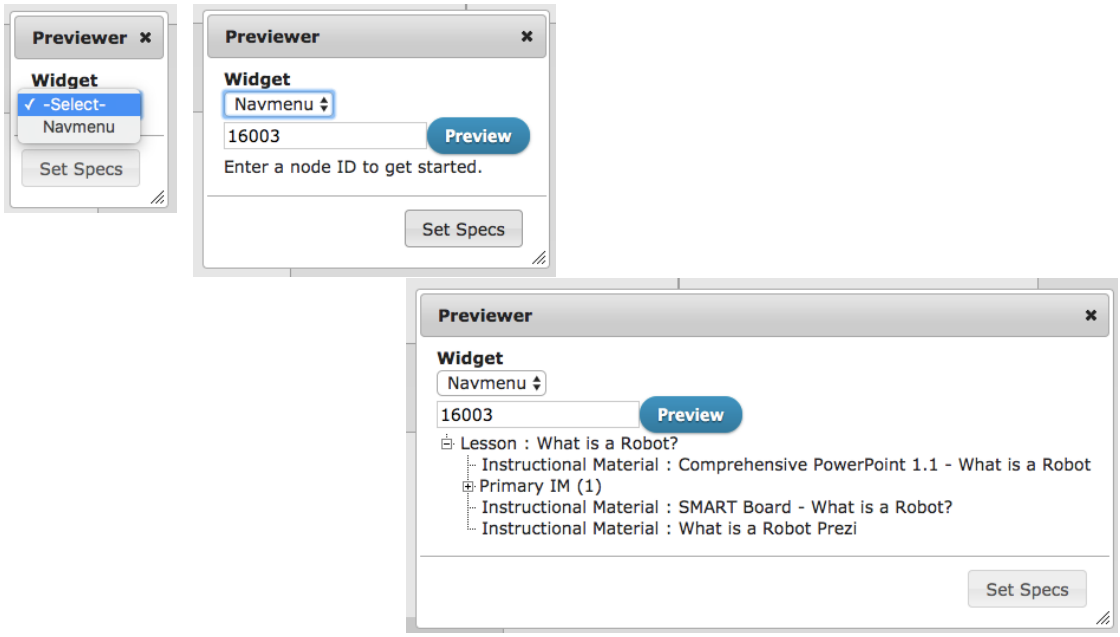


Figure 6.12: The widget previewer allows a user to preview their mappings in widgets in the system. First a widget is chosen, then the user chooses which content to preview (this field is pre-populated with a node id based on the selected mapping).

6.5 WIDGET SPECIFICATIONS

So far, all of our widgets have been created by a skilled developer. As part of that, the developer can tailor the functionality of the widgets to suit the needs of our non-technical users. We have also created a system where widgets can be implemented more flexibly, so that end-users can easily provide what we call a *specification* to control the details of how the widget works. Thus, widgets would be more accessible to non-technical users by enabling them to customize a widget as they perform schema mappings.

A widget specification allows a developer to parameterize parts of their widget for customization by users. For example, in the navigation widget in STEM-Robotics we may want to show the type of the resource in some cases, and the type of the relationship between the parent and child in other cases. In Figure 6.13

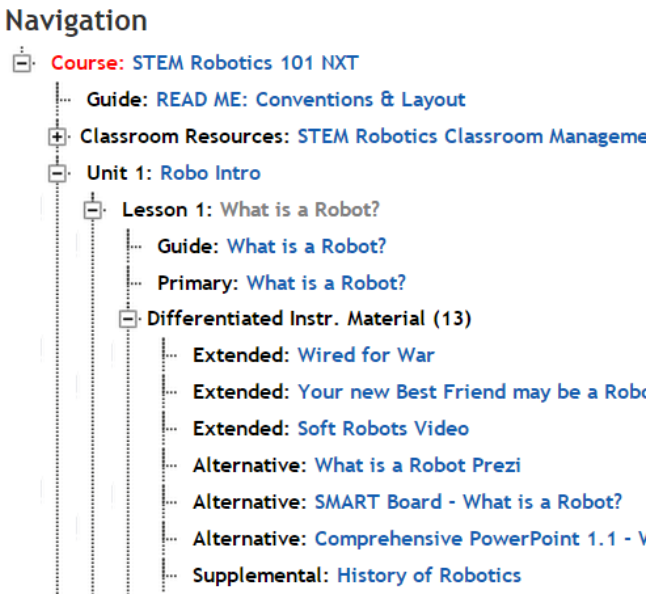


Figure 6.13: An instance of the naviation menu showing how different instance of educational materials may appear as "Primary" or "Differentiated".

the string showing the title of the instructional materials in the lesson is prefaced with "Primary" in one case and "Extended", "Alternative", or "Supplemental" (all relationship types) for other resources, even though all of the resources are of the same type. The different names come from the way in which the materials are associated with the lesson. The lesson's title is simply prefaced with its content type. In other applications, in contrast, the type may not be shown at all.

Defining and Configuring a Widget

A widget creator must create a default specification instance (by creating mapping-updatable parameters) that will be used as a starting point for the widget. Using only the default specification instance, the widget should work for any mapping and for any domain structure. In the case of the navigation tree above, parameters have been defined for "Cluster" (used to determine if an entry should be added between an entry and a subset of its children), "Color" (used to highlight the

title of an entry in a specific color), "Count" (a count of the children under this entry), "Title" (the text shown for the entry shown in blue in Figure 6.13), and "Type"(the text in the entry before the colon). The widget developer then also specifies a default value for these parameters. The parameters can then be updated while mapping, using the specification widget shown in Figure 6.14. This widget is launched from the preview widget shown in Figure 6.12 and already knows which mapping has been selected.

The default specifications are shown, in this case "NULL" for "Cluster", "Color", and "Count" which means that no entry will be added to the tree, it will not add any new color, and there will be no count. The "Title" and "Type" parameters are supplied by executable code, for "Title" we retrieve the node title from Drupal by accessing `$Node->Title` and we use the *type* operator on the node to determine the type.

The user can update parameters with either constant strings or executable code (by checking the "Executable?" box). For example, as shown in Figure 6.13, the user wants to add a level to the hierarchy in order to cluster specific resources together (e.g., differentiated instructional materials). The user also wants to show the counts under specific entries. The user has chosen to cluster resources accessed through this mapping in the "Diff IM: Alt" cluster. The user has also chosen to add a count by setting "Count" to "True". After adding this specification, the navigation tree is updated in the preview with the new cluster (shown in Figure 6.15). Originally, this type of widget modification would have required modifying the widget code. As the query interface was introduced and we sought to remove mapping-specific code from widgets, things like clusters could not be easily recreated without this kind of specification.

Widget specifications have allowed us to bring more end-user customization to widgets but the combination of executable and literal values in specifications means they can be confusing to users without knowledge of underlying Drupal

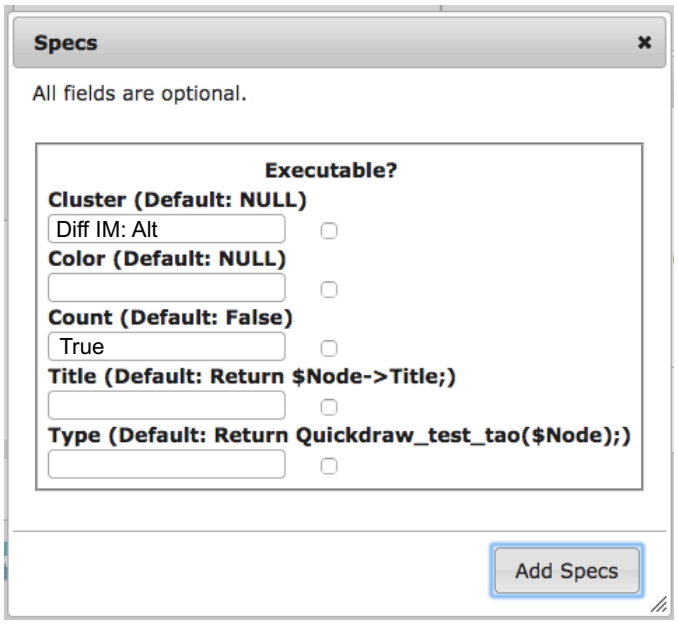


Figure 6.14: The mapping specification widget is shown for the mapping previewed in Figure 6.12.

concepts used in the executable parts of the specification.

6.6 POSTGRESQL

While the HERM-based formalism allowed us to precisely represent the complex data model of a web CMS, our implementation was still limited to the relational backend of the CMS, requiring the HERM to relational transformation to be written in the CMS code and the database, adding overhead to the query system. We strive to provide simple interfaces for both content authors for mapping and for widget developers when creating widgets, but the HERM-based query interface has the opposite effect, complicating widget development for developers accustomed to the relational model and SQL. As a result we decided to transition to the nested relational-based formalism presented in this thesis.

We decided to build our system using PostgreSQL because of its support for

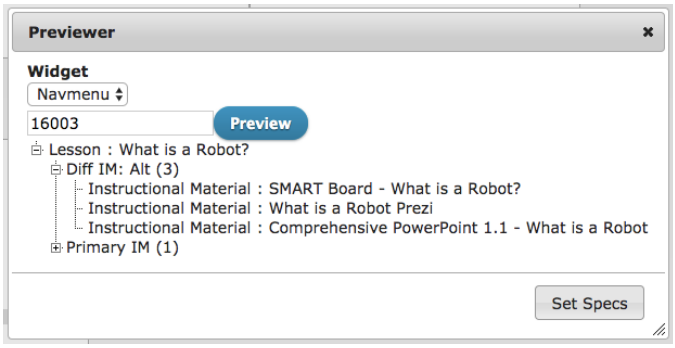


Figure 6.15: The navigation tree shown in Figure 6.12 is modified by the cluster created in the specification in Figure 6.14.

nested relational-like operations. The formal definitions of the local schema, domain structures, canonical structures, and mappings have all been stored as relations in the database. The full set of local radiance operators have been defined as PL/pgSQL [65] functions. These functions can then be used directly in SQL queries, as shown in the result sets in previous chapters. The functions directly implement the formalism defined in Chapter 3. For example, the *apply* operator is shown in Figure 6.16. The function follows Equation 3.1, finding mappings from the "dsa_mappings" function and then building results from the "Mapped" function.

This implementation provides an SQL interface using our operators, and through the use of the *apparent model* and *type* operators a widget developer need only know the relational model and not worry about the nested relational model.

6.7 CHAPTER SUMMARY

In this chapter we discussed the various implementations of local radiance that we have created. We have discussed the strengths and weakness of the various

```
CREATE FUNCTION alpha(dr text, dsaid text) RETURNS SETOF record
    LANGUAGE plpgsql
    AS $$
DECLARE
drm dr_lr_mrowtype;
ldb text;
result record;
un text;
BEGIN
un := '';
FOR result in
        SELECT r.ldbid, r.drm FROM dsa_mappings(dr,dsaid) r(ldbid text, drm dr_lr_mrowtype)
        LOOP
                IF un <> '' THEN
                        un := un || ' UNION ALL ' || Mapped(dr,result.ldbid, result.drm);
                ELSE
                        un := Mapped(dr,result.ldbid, result.drm);
                END IF;
        END LOOP;

        FOR result IN
                EXECUTE un
                LOOP
                return next result;
        END LOOP;
END;
$$;
```

Figure 6.16: The PostgreSQL implementation of the *apply* operator.

implementations. We have continued to build upon the strengths of each implementation while showing that even our earliest attempt at local radiance is still running and relevant.

Local radiance has been shown to work in multiple web frameworks and is continuing to be developed in a new framework. We have formalized our system using both HERM and the nested relational models and implemented both. We have made our work publicly available and have demonstrated the use of local radiance in four production websites since 2011 with over 4000 registered users and close to 430,000 page views in 2019.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this thesis we have presented our system for local radiance (LR). We described the formal foundations of local databases, domain and canonical structures, and mappings between them. We defined a base query language that can be used to create generic widgets that can radiate local relation and attribute names. We extended our query language with operators for local insert and update using queries from the domain and canonical levels. We have shown how our system can be used to perform more complex data-metadata transformations. We presented the evolution of system and formal ideas over time.

Below, we revisit the contributions made in this thesis to answer the research questions from Chapter 1 and discuss publications related to each.

How can we enable information integration that retains local beneficial heterogeneity?

Chapters 2 and 3 presented the formal definitions of our system and base query language showing how local database can be accessed from the domain and canonical levels while retaining the local beneficial heterogeneity.

In Chapter 2 we made the following contributions:

- We formally defined local databases, domain structures, and canonical structures.
- We defined our mapping system that allows mappings between local databases and domain structures; and, between domain structures and canonical structures.

- We defined the scope of mappings with our system and compare how our mappings compare to traditional tuple-generating dependencies, a common mechanism for database information integration.

In Chapter 3 we made the following contributions:

- We defined the apparent and underlying models used within our systems for query and storage.
- We defined the *apply* (α) operator that is introduced into queries at the domain level, which creates corresponding queries against local databases that return integrated data from all mapped local databases in the nested relational form of the domain structure.
- We defined the *canonical apply* (θ) operator that is introduced into queries at the canonical level, which creates corresponding queries against domain structures that return integrated data from all mapped domain structures in the nested relational form of the canonical structure (i.e., in the underlying model).
- We defined the *type* (τ) operator that provides local type information to the canonical or domain level.

In 2012, we presented generic widgets and end-user mapping at the International Conference on Conceptual Modeling (ER) 2012 [11]. In 2014, we presented our work on domain structures, mappings and the *apply* and *type* operators at the International Conference on Information Modeling and Knowledge Bases with the paper published in their journal [13]. In 2017, we presented our work on canonical structures in a chapter in Conceptual Modeling Perspectives [15].

How can we enable non-technical end-user schema mapping and information integration?

As noted above, Chapter 3 defined our mapping system and also made the following contribution:

- We evaluated the use of our mapping system by non-technical and technical users through a user study.

We showed that end-users do understand the mapping process and can create mappings successfully. In 2018, the results of our user study were presented in the Enterprise Modelling and Information Systems Architectures International Journal of Conceptual Modeling [18].

How can we build generic widgets that capture beneficial heterogeneity?

In Chapter 3, we showed how widgets can be built using the *apply*, *canonical apply*, and *type* operators. We also made the specific contributions:

- We defined the apparent and underlying models used within our systems for query and storage.

- We defined the *apparent model* (κ) operator, which provides a relational projection of the underlying model of a canonical or domain structure into the apparent model.

By providing a relational model, we provide access to our system in a way that is already understood and used by most widget developers. We presented the use of LR widgets in a digital library setting in 2012 [16, 19] and 2013 [10, 17, 21]. We showed how these widgets can used to facilitate educators at the ACM Conference on Computer Science Education [20].

Can we leverage local radiance to create generic local data creation and manipulation widgets?

Chapter 4 presented our extensions to our base query language making the following contributions:

- We defined the *local document* operator (β) that, given a domain relation, will return a document for every tuple in the result of an *apply* operator on that domain relation. Each returned tuple contains the schema and data for all attributes from the local relation that corresponds to the mapped tuple.

- We defined the *empty document* operator (ϵ) that, given a domain relation, will return an empty document in the schema of each local relation that has been mapped to the domain relation.

- We defined *insert* and *update* operators that use β and ϵ to insert and update local data from the domain level.

- We showed how the operators can be used for cloning and exploration in a digital repository.

We presented our work on the *local document* operators at ER 2014 [14].

Can we empower end-users to perform complex data transformation tasks and widget customization?

In Chapter 5, we showed how our system can be extended to perform data-to-metadata transformation and made the following contributions:

- We showed how correspondences between domain structures and local schemas can support data-to-metadata transformations.

- We presented a case study that shows a complex, faceted browse widget in a digital library that uses data-to-metadata transformation.

- We extended our simple correspondences to include a predicate in order to support the classical database pivot operation.

- We compared our system against similar systems that perform data-to-metadata transformations.

In 2016, we presented our work on pivot and unpivot at ER 2016 [12].

What is the best way to formalize and implement an LR system? Chapter 6 presented the evolution of the LR system. We discussed the limitations of our previous HERM-based system and Chapters 2 and 3 presented our current nested relational-based system. Chapter 3 made the following contributions:

- We defined relational equivalences that can be used with our operators and showed how they can be used to optimize performance in our system.
- We evaluated the performance of our system against hand written integration queries as well as custom-coded in a web development framework (Drupal [33]).

By using the nested relational model, we were able to leverage existing relational equivalences and optimizations and more easily integrate our framework into existing CMSs.

7.1 FUTURE WORK

We have shown that content authors can and will create simple mappings that can enable a large variety of generic widgets. We see avenues of extending both the mapping research and the widgets in this work. We conclude with a discussion of possible areas of future research.

7.1.1 Join-Path Mappings

We have purposely limited our system to simple relation-to-relation mappings in order to facilitate non-technical users. A logical next step for future research is to expand our mappings to more complex forms.

A common application for more complex mappings is the existing tree-based navigation widget that we already have. For example, say we would like to have

an instance of the widget that only shows leaf-level instructional materials from all of the courses. If a mapping could encompass the path of joins in a local schema (such as course to unit to lesson to instructional material) then the widget would work as normal.

There are two possible avenues to explore this. First, if we only allow foreign-key-based joins, then very little of our existing infrastructure need change, as we could use the combination of mapping and correspondence ids as well as the foreign key relationships in the local schema.

Second, a more generalized mapping could be created that contains not only the mapping and correspondence ids in the "meta" nested relation of the underlying domain or canonical attributes, but we could also move the local ids from the domain or canonical id attribute into the "meta" nested relation. But with a more complete set of data in the underlying model, a mapping could become increasingly complex. This change to the mapping system would also require a redefinition of the *apply* and *canonical apply* operators. Any increase in the expressive power of the mappings could also have the negative consequence of making mapping more difficult for end-users.

7.1.2 Enhancing and Extending Local Radiance Infrastructure

Our mapping study showed that non-technical users can perform the schema mapping tasks necessary to use our system. But the user interface used in our study does require users to understand or infer some of the underlying structure of the Drupal CMS. One of our goals is to make mappings in our system as easy as drawing lines (like Clio [57]) or highlighting relevant parts of a web page.

We plan to build an easier-to-use mapping interface that can produce mappings for any LR-enabled CMS. We envision an interface where content authors can specify mappings while viewing their own data on their web-page.

We also propose to define an exchange format for our mappings. By exchanging

and sharing mappings, we can enable widgets that work across applications and we can also start to collect mappings and use them together to perform more complex integration or reasoning tasks (discussed below).

We envision a central portal that will host the mapping interface, domain structures, canonical structures, and widgets. The local web CMS would require a small amount of infrastructure to be able to correctly send local database information to the portal. The portal would then supply embeddable widgets (in the same fashion as maps and videos are currently embeddable from their host sites into other sites). The use of a portal allows the possibility to extend LR to plain websites (like a superimposed information application [52]).

7.1.3 Reasoning Over Mappings and Semantic Web Integration

Once a portal is created and all mappings are hosted centrally, we can gain the ability to glean extra information from the collected set of mappings. As users generate mappings to our structures, we can then use these mappings to provide semantic web integration capabilities similar to federated databases [76] or data warehouses [41].

While we have focused on widgets that enable functionality for local content authors, we can also build widgets for the semantic web that uses the mappings created by content authors. Gangemi and Presutti [37] identify the "knowledge boundary problem" as the problem of identifying meaningful units within the semantic web. Our domain structures are precisely the kind of meaningful units that they seek to make explicit. Widgets can be built that generate RDF [71] and OWL [84] data using the concepts and semantics represented in the domain structures.

While many solutions for semantic-web-based integration have been proposed and developed and been successful, generating good mappings for large systems manually is difficult [69]. We believe that our use of small domain fragments

that are mapped by the content authors will lead to high quality mappings. Over time, the collected set of mappings for each domain structure can provide an array of synonyms for entities and relationships. Our mappings, in the context of the meaningful units of our domain structures, will hopefully contribute to the creation of better semantic web ontologies.

REFERENCES

[1] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM J. Comput.*, 8(2):218–246, 1979.

[2] J. Q. Anderson and H. Rainie. *The fate of the Semantic Web*. Pew Internet & American Life Project, 2010.

[3] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to non-relational database systems: the sos platform. In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering*, CAiSE'12, pages 160–174, Gdańsk, Poland. Springer-Verlag, 2012.

[4] P. Atzeni, G. Mecca, and P. Merialdo. Managing web-based data: database models and transformations. *IEEE Internet Computing*, 6(4):33–37, July 2002.

[5] P. Atzeni, P. Merialdo, and G. Mecca. Data-intensive web sites: design and maintenance. *World Wide Web*, 4(1-2):21–47, Oct. 2001.

[6] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[7] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *SIGMOD Rec.*, 29(4):55–63, Dec. 2000.

[8] M. Blaha. *Patterns of Data Modeling*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2010.

[9] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, Chicago, IL, USA. ACM, 2006.

[10] S. Britell and L. Delcambre. Checking out: customizing and downloading complex and compound digital library resources. In *Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '13, pages 379–380, Indianapolis, Indiana, USA. ACM, 2013.

[11] S. Britell and L. M. L. Delcambre. Mapping semantic widgets to web-based, domain-specific collections. In *Proceedings of the 31st International Conference on Conceptual Modeling*, ER'12, pages 204–213, Florence, Italy. Springer-Verlag, 2012.

[12] S. Britell, L. M. L. Delcambre, and P. Atzeni. Facilitating data-metadata transformation by domain specialists in a web-based information system using simple correspondences. In *Proceedings of the 35rd International Conference on Conceptual Modeling*, ER'16, pages 445–459, Gifu, Japan. Springer-Verlag, 2016.

[13] S. Britell, L. M. L. Delcambre, and P. Atzeni. Flexible Information Integration with Local Dominance. *Information Modelling and Knowledge Bases*, XXVI:21–40, 2014.

[14] S. Britell, L. M. L. Delcambre, and P. Atzeni. Generic data manipulation in a mixed global/local conceptual model. In E. Yu, G. Dobbie, M. Jarke, and S. Purao, editors, *Proceedings of the 33rd International Conference on Conceptual Modeling*, ER'14, pages 246–259, Atlanta, GA, USA. Springer-Verlag, 2014.

[15] S. Britell, L. M. L. Delcambre, and P. Atzeni. *Web system development using polymorphic widgets and generic schemas*. In *Conceptual Modeling Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2017, pages 121–135.

[16] S. Britell, L. M. L. Delcambre, L. N. Cassel, E. A. Fox, and R. Furuta. Enhancing digital libraries and portals with canonical structures for complex objects. In *Proceedings of the Second International Conference on Theory and Practice of Digital Libraries*, TPDL'12, pages 420–425, Paphos, Cyprus. Springer-Verlag, 2012.

[17] S. Britell, L. M. L. Delcambre, L. N. Cassel, and R. Furuta. Checking out: download and digital library exchange for complex objects. In T. Aalberg, C. Papatheodorou, M. Dobreva, G. Tsakonas, and C. J. Farrugia, editors, *Research and Advanced Technology for Digital Libraries*, pages 48–59, Berlin, Heidelberg. Springer Berlin Heidelberg, 2013.

[18] S. Britell and L. M. Delcambre. Evaluating user behavior as they create mappings in a web development system using local radiance. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 13:234–242, 2018.

[19] S. Britell, L. Delcambre, L. Cassel, E. Fox, and R. Furuta. Exploiting canonical structures to transmit complex objects from a digital library to a portal. In *Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries*, JCDL '12, pages 377–378, Washington, DC, USA. ACM, 2012.

[20] S. Britell, L. Delcambre, E. Fox, and R. Steele. Curriculum collaboration, customization, and reuse: creating communities in digital repositories (abstract only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 731–731, Denver, Colorado, USA. ACM, 2013.

[21] P. Brusilovsky, Y. Lin, C. Wongchokprasitti, S. Britell, L. M. L. Delcambre, R. Furuta, K. Chiluka, L. N. Cassel, and E. Fox. Social navigation support

for groups in a community-based educational portal. In T. Aalberg, C. Papatheodorou, M. Dobreva, G. Tsakonas, and C. J. Farrugia, editors, *Research and Advanced Technology for Digital Libraries*, pages 429–433, Berlin, Heidelberg. Springer Berlin Heidelberg, 2013.

[22] A. Calil and R. dos Santos Mello. Simplesql: a relational layer for simpledb. In *Proceedings of the 16th East European Conference on Advances in Databases and Information Systems*, ADBIS'12, pages 99–110, Poznań, Poland. Springer-Verlag, 2012.

[23] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[24] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Comput. Netw.*, 33(1-6):137–157, June 2000.

[25] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

[26] L. S. Colby. A recursive algebra and query optimization for nested relations. *SIGMOD Rec.*, 18(2):273–283, June 1989.

[27] C# — Microsoft Docs. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/csharp` (visited on 09/02/2019).

[28] C. Cunningham, C. A. Galindo-Legaria, and G. Graefe. Pivot and unpivot: optimization and execution strategies in an rdbms. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 998–1009, Toronto, Canada. VLDB Endowment, 2004.

[29] A. Das Sarma, X. Dong, and A. Halevy. Bootstrapping pay-as-you-go data integration systems. In *Proceedings of the 2008 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '08, pages 861–874, Vancouver, Canada. ACM, 2008.

[30] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, Sept. 1982.

[31] L. M. L. Delcambre, D. Maier, R. Reddy, and L. Anderson. Structured Maps: modeling explicit semantics over a universe of information. *International Journal on Digital Libraries*, 1(1):20–35, Apr. 1997.

[32] S. Dessloch, M. A. Hernandez, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: integrating schema mapping and etl. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1307–1316, Washington, DC, USA. IEEE Computer Society, 2008.

[33] Drupal. URL: `http://drupal.org` (visited on 09/02/2019).

[34] A. Eisenberg and J. Melton. Sql/xml is making good progress. *SIGMOD Rec.*, 31(2):101–108, June 2002.

[35] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer Publishing Company, Incorporated, 2nd edition, 2013.

[36] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, Oct. 1982.

[37] A. Gangemi and V. Presutti. Towards a pattern science for the semantic web. *Semant. web*, 1(1,2):61–68, Apr. 2010.

[38] Gleaning Resource Descriptions from Dialects of Languages (GRDDL). URL: `http://www.w3.org/2004/01/rdxh/spec` (visited on 09/02/2019).

[39] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: a powerful approach to aggregation. In number 1995-32. Stanford InfoLab, 1995.

[40] M. A. Hernández, P. Papotti, and W.-C. Tan. Data exchange with data-metadata translations. *Proc. VLDB Endow.*, 1(1):260–273, Aug. 2008.

[41] W. H. Inmon. *Building the Data Warehouse,3rd Edition*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 2002.

[42] Internet Live Stats - Total number of Websites. URL: `https://www.internetlivestats.com/total-number-of-websites/` (visited on 09/02/2019).

[43] Java 8. URL: `https://java.com/en/download/faq/java8.xml` (visited on 09/02/2019).

[44] C. Jones. End-user programming. *Computer*, 28(9):68–70, Sept. 1995.

[45] JavaScript Object Notation. URL: `https://www.json.org/` (visited on 09/02/2019).

[46] A. M. Keller, R. Jensen, and S. Agarwal. Persistence software: bridging object-oriented programming and relational databases. *SIGMOD Rec.*, 22(2):523–528, June 1993.

[47] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: an extension to sql for multidatabase interoperability. *ACM Trans. Database Syst.*, 26(4):476–519, Dec. 2001.

[48] M. Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 233–246, Madison, Wisconsin. ACM, 2002.

[49] M. Levene and G. Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, Berlin, Heidelberg, 1999.

[50] H. Lieberman, F. Paternò, M. Klann, and V. Wulf. *End-user development: an emerging paradigm*. In *End User Development*. H. Lieberman, F. Paternò, and V. Wulf, editors. Springer Netherlands, Dordrecht, 2006, pages 1–8.

[51] H. Ma, R. Noack, K.-D. Schewe, and B. Thalheim. Using Meta-Structures in Database Design. *Informatica*, 34(3):387–403, 2010.

[52] D. Maier and L. M. L. Delcambre. Superimposed information for the internet. In *ACM SIGMOD Workshop on The Web and Databases, WebDB 1999, Philadelphia, Pennsylvania, USA, June 3-4, 1999. Informal Proceedings*, pages 1–9, 1999.

[53] E. Mäkelä, K. Viljanen, O. Alm, J. Tuominen, O. Valkeapää, T. Kauppinen, J. Kurki, R. Sinkkilä, T. Känsälä, R. Lindroos, et al. Enabling the semantic web with ready-to-use web widgets. In *Proceedings of the First International Conference on Industrial Results of Semantic Technologies-Volume 293*, pages 56–69. CEUR-WS. org, 2007.

[54] P. Merialdo, P. Atzeni, and G. Mecca. Design and development of data-intensive web sites: the araneus approach. *ACM Trans. Internet Technol.*, 3(1):49–92, Feb. 2003.

[55] Microformats. URL: `http://microformats.org/wiki/Main_Page` (visited on 09/02/2019).

[56] Microsoft SQL Server. URL: `https://docs.microsoft.com/en-us/sql/?view=sql-server-2017` (visited on 09/02/2019).

[57] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. Howard Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIGMOD Rec.*, 30(1):78–83, Mar. 2001.

[58] S. Murthy, D. Maier, and L. Delcambre. Mash-o-matic. In *Proceedings of the 2006 ACM Symposium on Document Engineering*, DocEng '06, pages 205–214, Amsterdam, The Netherlands. ACM, 2006.

[59] J. Nielsen. Usability inspection methods. In *Conference Companion on Human Factors in Computing Systems*, CHI '94, pages 413–414, Boston, Massachusetts, USA. ACM, 1994.

[60] B. Nowack. Paggr: linked data widgets and dashboards. *Web Semant.*, 7(4):272–277, Dec. 2009.

[61] N. F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, Dec. 2004.

[62] A. Olivé. *Conceptual Modeling of Information Systems*. Springer-Verlag, Berlin, Heidelberg, 2007.

[63] Oracle Database. URL: `https://docs.oracle.com/en/database/oracle/oracle-database/index.html` (visited on 09/02/2019).

[64] ORACLE-BASE - SQL/XML (SQLX) : Generating XML using SQL. URL: `https://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql` (visited on 09/02/2019).

[65] PostgreSQL: Documentation: 11: Chapter 43. PL/pgSQL - SQL Procedural Language. URL: `https://www.postgresql.org/docs/current/plpgsql.html` (visited on 09/02/2019).

[66] PostgreSQL: Documentation: 11: 9.14. XML Functions. URL: `https://www.postgresql.org/docs/current/functions-xml.html` (visited on 09/02/2019).

[67] V. Presutti and A. Gangemi. Content ontology design patterns as practical building blocks for web ontologies. In *Proceedings of the 27th International Conference on Conceptual Modeling*, ER '08, pages 128–141, Barcelona, Spain. Springer-Verlag, 2008.

[68] pureXML. URL: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.xml.doc/doc/c0022308.html (visited on 09/02/2019).

[69] E. Rahm. *Towards large-scale schema and ontology matching.* In *Schema Matching and Mapping.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pages 3–27.

[70] RDFa. URL: http://www.w3.org/TR/rdfa-syntax/ (visited on 09/02/2019).

[71] Resource Description Framework. URL: http://www.w3.org/RDF/ (visited on 09/02/2019).

[72] J. Rode, Y. Bhardwaj, M. A. Pérez-Quiñones, M. B. Rosson, and J. Howarth. As easy as "click": end-user web engineering. In *Proceedings of the 5th International Conference on Web Engineering*, ICWE'05, pages 478–488, Sydney, Australia. Springer-Verlag, 2005.

[73] H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, Apr. 1986.

[74] Schema.org. URL: http://schema.org (visited on 09/02/2019).

[75] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, May 2006.

[76] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, Sept. 1990.

[77] SPARQL Query Language for RDF. URL: http://www.w3.org/TR/rdf-sparql-query/ (visited on 09/02/2019).

[78] J. F. Terwilliger, L. M. L. Delcambre, D. Maier, J. Steinhauer, and S. Britell. Updatable and evolvable transforms for virtual databases. *Proc. VLDB Endow.*, 3(1-2):309–319, Sept. 2010.

[79] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2000.

[80] B. Thalheim, K.-D. Schewe, and H. Ma. Conceptual application domain modelling. In *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, APCCM '09, pages 49–58, Wellington, New Zealand. Australian Computer Society, Inc., 2009.

[81] Topic Maps. URL: `http://www.topicmaps.org/` (visited on 09/02/2019).

[82] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira. An Effective Scalable SQL Engine for NoSQL Databases. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 155–168. Springer, Springer, Berlin, Heidelberg, 2013.

[83] W3Tech - Usage Statistics and Market Share of Content Management Systems, October 2019. URL: `https://w3techs.com/technologies/overview/content_management/all` (visited on 09/02/2019).

[84] Web Ontology Language OWL. URL: `http://www.w3.org/2004/OWL/` (visited on 09/02/2019).

[85] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1435–1444, San Jose, California, USA. ACM, 2007.

[86] C. M. Wyss and E. L. Robertson. A formal characterization of pivot/unpivot. In *Proceedings of the 14th ACM International Conference on Information*

and Knowledge Management, CIKM '05, pages 602–608, Bremen, Germany. ACM, 2005.

[87] C. M. Wyss and E. L. Robertson. Relational languages for metadata integration. *ACM Trans. Database Syst.*, 30(2):624–660, June 2005.

[88] Extensible Markup Language (XML). URL: `https://www.w3.org/XML/` (visited on 09/02/2019).

[89] XML Path Language (XPath). URL: `https://www.w3.org/TR/1999/REC-xpath-19991116/` (visited on 09/02/2019).

[90] The Official YAML Web Site. URL: `https://yaml.org/` (visited on 09/02/2019).

Appendix A

CANONICAL VERSIONS OF LOCAL INSERT AND UPDATE OPERATORS

$$\beta(cr, csaid) = \bigcup_{mid \in \textbf{mids}(cr, csaid)} \textbf{build\_local}(mid) \tag{1}$$

$$\textbf{mids}(cr, csaid) = \pi_{\textbf{SPLIT}(id,\text{`.'})[2]}(\theta(cr, csaid)) \tag{2}$$

$$\textbf{build\_local}(mid) = \nu_{mid,ldb,lr,lrkey,attrs:local\_doc}\left(\nu_{lrkeyattr,lrkeyvalvalue:lrkey}\left(\nu_{name,value:attrs}\left(\bigcup_{\substack{(ldbid,lr,lrkey,attr)\in \\ \textbf{local\_from\_mid}(mid)}} \left(\pi_{\substack{\text{“}[ldbid]||.||[lr]||.||[mid]||.||[lrkey]\text{”}\rightarrow id, \\ \text{“}[mid]\text{”}\rightarrow mid, \text{“}[ldbid]\text{”}\rightarrow ldb, \text{“}[lr]\text{”}\rightarrow lr, \\ \text{“}[lrkey]\text{”}\rightarrow lrkeyattr, [lrkey]\rightarrow lrkeyval \\ \text{“}[attr]\text{”}\rightarrow name, \\ [attr]\rightarrow value}} (\textbf{table\_scan}(ldbid, lr))\right)\right)\right)\right) \tag{3}$$

$$\textbf{local\_from\_mid}(mid) = \pi_{\substack{ldb.id\rightarrow ldbid, \\ ldb.lrs.name\rightarrow lr, \\ ldb.lrs.key\rightarrow lrkey, \\ ldb.lrs.attrs.name\rightarrow attr}} \left((\sigma_{ds\_ldb\_m.dr\_lr\_ms.id=mid}(ds\_ldb\_m)) \bowtie_{\substack{ds\_ldb\_m.ldbid=ldb.id \wedge \\ ds\_ldb\_m.dr\_lr\_ms.lr=ldb.lrs.name}} ldb\right) \tag{4}$$

Where the **table_scan**($ldbid, lr$) function performs a table scan operation on the local relation lr in the local database $ldbid$; and, the **SPLIT**(s,d) function splits a string (s) on a delimiter (d) and returns an array of the resulting substrings.

Equation A.1: Local Document Operator (β)

$$\epsilon(cr, csaid) = \bigcup_{mid \in \textbf{mids}(cr, csaid)} \textbf{build\_empty\_local}(mid) \tag{1}$$

$$\textbf{mids}(cr, csaid) = \pi_{\textbf{SPLIT}(id,'.')[2]}(\theta(cr, csaid)) \tag{2}$$

$$\textbf{build\_empty\_local}(mid) = \nu_{ldb,lr,attrs:local\_doc}\left(\nu_{lrkeyattr,lrkeyvalvalue:lrkey}\left(\nu_{name,value:attrs}\left(\bigcup_{\substack{(ldbid,lr,lrkey,attr)\in \\ \textbf{local\_from\_mid}(mid)}} \underset{\rightarrow(id,mid,ldb,lr,lrkeyattr,lrkeyval,name,value)}{(NULL,[mid],[ldbid],[lr],[lrkey],NULL,[attr],NULL)}\right)\right)\right) \tag{3}$$

$$\textbf{local\_from\_mid}(mid) = \pi_{\substack{ldb.id\rightarrow ldbid, \\ ldb.lrs.name\rightarrow lr, \\ ldb.lrs.key\rightarrow lrkey, \\ ldb.lrs.attrs.name\rightarrow attr}}\left((\sigma_{ds\_ldb\_m.dr\_lr\_ms.id=mid}(ds\_ldb\_m)) \underset{\substack{ds\_ldb\_m.ldbid=ldb.id \wedge \\ ds\_ldb\_m.dr\_lr\_ms.lr=ldb.lrs.name}}{\bowtie} ldb\right) \tag{4}$$

Where the **table_scan**$(ldbid, lr)$ function performs a table scan operation on the local relation lr in the local database $ldbid$; and, the **SPLIT**(s,d) function splits a string (s) on a delimiter (d) and returns an array of the resulting substrings.

Equation A.2: Empty Document Operator (ϵ)