Incorporating Relational Data into the Semantic Web

Inaugural-Dissertation
zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von
Cristian Pérez de Laborda Schwankhart
aus Bilbao, Spanien

August 2006

Aus dem Institut für Informatik
der Mathematisch-Naturwissenschaftlichen Fakultät

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Stefan Conrad
Koreferent: Prof. Dr.-Ing. Kai-Uwe Sattler (TU Ilmenau)
Tag der mündlichen Prüfung: 21.11.2006

"*[The Semantic Web] is about the data which currently is in relational databases, XML documents, spreadsheets, and proprietary format data files, and all of which would be useful to have access to as one huge database.*" [Upd05]

Tim Berners-Lee
Director of the W3C

Acknowledgements

This thesis is the result of three and a half years of work as a research and teaching assistant at the Databases and Information Systems group at the University of Düsseldorf. I have to express my gratitude to all the people who made this work possible.

First of all, I like to express my warmest thanks to my supervisor Prof. Dr. Stefan Conrad for giving me the opportunity to join his research group and for encouraging me to realize my ideas on the Semantic Web. I also like to thank the second reviewer of this thesis Prof. Dr.-Ing. Kai-Uwe Sattler for his interest in my work and the time for reading and commenting this thesis.

This work would not have been possible without the cooperation and support of my colleagues and friends at the Institute of Computer Science, who created a wonderful atmosphere. I thank Dr. Christopher Popfinger, my longtime fellow student for the joint studies, research, and activities performed during the last years in Düsseldorf and Munich. I also thank Dr. Evguenia Altareva, especially for her patience in sharing an office with me. Additionally, I extend my compliments to Johanna Vompras, Tobias Riege, Christian Lochert, and all the remaining people who helped me to have a great time here in Düsseldorf doing no research.

Guido Königstein and Marga Potthoff supported the realization of this thesis through their background help. I would probably have failed without them because of technical or administrative barriers.

I am also grateful to all the students who helped to mature and realize my ideas through their work and questions, especially to Yves Thielan, Matthäus Zloch, and Przemysław Dzikowski.

Last but not least, I want to dedicate this thesis to my whole family, especially to my parents, who continuously supported and encouraged me from the very beginning. I would not have achieved anything without them.

Düsseldorf, August 2006 Cristian Pérez de Laborda Schwankhart

Abstract

The aim of the Semantic Web as promoted by the World Wide Web Consortium is to enable software agents to access distributed information and to apply inference rules, so that new knowledge can be deduced. Nevertheless, since the vast majority of data is still modeled and stored in relational databases, this information is out of reach for most Semantic Web applications. Consequently, local users usually create their own manual relational to semantic mappings or convert it in a manual, time-consuming, and error-prone process into a corresponding Semantic Web representation.

In this thesis we present Relational.OWL, a technique to automatically convert a relational database into a Semantic Web representation, enabling Semantic Web applications to access data actually stored in relational databases using their own built-in functionality. Since the Relational.OWL representation of the database does not result in objects containing real semantics, we additionally show how to create mappings from the relational model to a target ontology using an arbitrary closed RDF query language.

Using current Semantic Web techniques, a formerly relational data item, once converted to its Semantic Web representation, can neither be identified unambiguously, nor be backtracked to its original storage location in the relational database. We hence introduce the novel URI scheme `db` for identifying not only databases, but also their schema and data components like tables or columns, giving us the possibility to specify the exact and identifying storage location of any data item in its original data source.

Since sharing information in distributed environments like the Semantic Web often leads to recurring problems, we finally present the Link Pattern Catalog as a modeling guideline for problems appearing during the design and implementation of such information sharing environments.

Contents

1 Motivation **1**
- 1.1 Contributions . . . 3
- 1.2 Outline . . . 4

2 Background **7**
- 2.1 Relational Databases . . . 7
  - 2.1.1 Relational Model . . . 8
    - 2.1.1.1 Relational Data Model . . . 8
    - 2.1.1.2 Relational Algebra . . . 9
  - 2.1.2 SQL . . . 11
- 2.2 The Semantic Web . . . 16
  - 2.2.1 Introduction . . . 16
  - 2.2.2 Semantic Web Technologies . . . 17
    - 2.2.2.1 RDF . . . 17
    - 2.2.2.2 OWL Web Ontology Language . . . 22
    - 2.2.2.3 RDF Query Languages . . . 23

3 Bridging the Semantic Gap **27**
- 3.1 Relational.OWL . . . 28
  - 3.1.1 Motivation . . . 29
    - 3.1.1.1 Data for the Semantic Web . . . 29
    - 3.1.1.2 Relational.OWL as an Exchange Format . . . 30
    - 3.1.1.3 Peer-to-Peer Databases . . . 32
  - 3.1.2 Relevant Metadata . . . 34
  - 3.1.3 The Relational.OWL Ontology . . . 35
  - 3.1.4 Schema Representation . . . 37
  - 3.1.5 Data Representation . . . 38
  - 3.1.6 Data Overhead in the Data Exchange Process . . . 40
- 3.2 Relational.OWL and RDF Query Languages . . . 41
  - 3.2.1 Relational.OWL and RDQL . . . 42
    - 3.2.1.1 Selection . . . 43
    - 3.2.1.2 Projection . . . 44
    - 3.2.1.3 Set Union . . . 45

3.2.1.4 Set Difference . . . 45
3.2.1.5 Cartesian Product . . . 46
3.2.1.6 (Equi-)Join . . . 47
3.2.1.7 Discussion . . . 48
3.2.2 Relational.OWL and SPARQL . . . 49
3.2.2.1 Selection . . . 49
3.2.2.2 Projection . . . 50
3.2.2.3 Set Union . . . 51
3.2.2.4 Set Difference . . . 52
3.2.2.5 Cartesian Product . . . 53
3.2.2.6 (Equi-)Join . . . 54
3.2.2.7 Discussion . . . 55
3.3 Relational to Semantic Mapping . . . 56
3.3.1 Requirements . . . 56
3.3.2 Definitions . . . 57
3.3.3 Mapping Process . . . 58
3.3.4 Characteristics . . . 59
3.3.5 Classification . . . 60
3.3.6 Sample Mapping . . . 61
3.3.7 Evaluation . . . 62
3.4 Related Work . . . 63
3.5 Discussion and Future Work . . . 65

4 A Novel URI for Databases 67
4.1 Motivation . . . 67
4.2 Challenges Designing an Identifier . . . 69
4.3 Model . . . 70
4.4 Example . . . 72
4.4.1 XML Data Exchange . . . 73
4.4.2 Relational.OWL Representation of a Database . . . 74
4.5 Related Work . . . 77
4.6 Discussion and Future Work . . . 77

5 Applications 79
5.1 Relational.OWL Implementations . . . 79
5.1.1 Relational.OWL Application . . . 80
5.1.1.1 Introduction . . . 80
5.1.1.2 Usage . . . 81
5.1.2 Relational.OWL with XSLT and XQuery . . . 84
5.1.2.1 Introduction . . . 84
5.1.2.2 Implementation . . . 85
5.2 RDQuery . . . 86
5.2.1 Introduction . . . 86

5.2.2 Query Translation . . . 87
5.2.3 Usage . . . 89
5.3 DÍGAME . . . 90
5.3.1 Motivation . . . 90
5.3.2 DÍGAME Architecture . . . 92
5.3.2.1 Basic Functionality . . . 92
5.3.2.2 Components of the Architecture . . . 94
5.3.3 Characteristics . . . 96
5.3.4 DÍGAME System Design . . . 99
5.3.5 Related Work . . . 100
5.3.6 Discussion and Future Work . . . 102

6 Link Patterns 105
6.1 Introduction . . . 105
6.2 The Data Link Modeling Language (DLML) . . . 106
6.2.1 Motivation . . . 106
6.2.2 Components . . . 107
6.2.3 Example . . . 109
6.3 Link Patterns . . . 110
6.3.1 Elements of a Link Pattern . . . 110
6.3.2 Classification . . . 111
6.3.3 Usage . . . 112
6.4 Link Pattern Catalog . . . 113
6.4.1 Elementary Link Patterns . . . 114
6.4.2 Data Independent Link Patterns . . . 115
6.4.3 Data Sensitive Link Patterns . . . 116
6.5 Example . . . 118
6.6 Related Work . . . 120
6.7 Discussion and Future Work . . . 120

7 Conclusion 123

Bibliography 125

List of Figures 144

List of Tables 145

Chapter 1

Motivation

With billions of static pages, the World Wide Web (WWW) is probably the largest information repository of the world. To this, we have to add the data of the deep or hidden web [Ber00], i.e. information usually stored in relational databases, made accessible through online forms or dynamic web pages. As a result, finding the right pages becomes a challenging task, which has to be mastered by anybody searching for specific information on the Web. Although search engine algorithms become better and better, the user still faces a massive information overload, where he has to decide on his own, which information is relevant and which is not. A search engine is only able to assist the user during this time-consuming process, since most of data stored on web servers, although being machine readable, is not machine understandable.

Realizing this situation, the World Wide Web Consortium (W3C) with Tim Berners-Lee as its director, started to promote their vision of the *Semantic Web* (cf. [BL98b, BLHL01, BLM02]). This next generation Web provides software agents machine processable and understandable data. As a consequence, software agents process and interpret such information on their own and are able to identify relevant data. Depending on the intended functionality, those agents can either present the collected information to a user or deduce new knowledge on their own using inference rules.

Since its very beginning in the late 1990ies, the Semantic Web and its knowledge representation techniques have produced big impacts in several research fields, ranging from Peer-to-Peer (P2P) systems over Life Sciences to the annotation of multimedia data.

Peer-to-Peer Systems: The characteristic feature of a P2P-based system is the extensive autonomy of the participating nodes. This autonomy does not only affect the connectivity to the net, but also data storage, data representation, and exchange formats. Indeed, interacting nodes need a common language, i.e. exchange format, on which they can rely, neverthe-

less, it is not guaranteed that both peers interpret the same information equally.

Due to the short availability of potentially any peer, the negotiation of data and meta data representation formats, including the exchange protocol, becomes a challenging task. An exchange format, which can be understood instantly by all exchange partners would be more useful. Based on a common semantic data representation, participants on P2P systems can map their data to commonly available concepts, enabling the remaining peers to know which information is stored on which peers, how to create mappings among these storage locations, and how to access it using a common query language (cf. e.g. [SS06], Section 3.1.1.3, and Section 5.3).

Multimedia Data: Searching for specific pictures in huge image banks is a difficult tasks. On the one hand, these pictures have to be annotated manually with corresponding keywords, which on the other hand have to be used by the user to explicitly find this specific image. Typically, different annotators use different terms for describing the same thing, e.g. a skyscraper could be annotated using the term *real estate* or by *building*, depending on the vocabulary or the background of the annotating person.

With this kind of keyword-based annotation, the Semantic Web can support content-based image retrieval. Using ontologies as shared terminologies for the annotation of images, the semantics of a picture is not only described, but made comparable and interpretable (cf. e.g. [SDWW01, HSWW03]).

Life Sciences: Being a data intensive research field, the so-called Life Sciences have soon recognized the possibilities resulting from the combination of their research area with knowledge representation techniques as supported by the Semantic Web. Similar to the annotation of multimedia data, large companies or organizations like Pfizer, or the National Cancer Institute (NCI) hope to integrate existing research information using the Semantic Web to improve the knowledge management within their institution (cf. [HdCD+05, DRR+06]). Additionally, health care centers in the US already have introduced clinical decision support systems, which based on Semantic Web techniques like inference rules and ontologies, compute clinical guidelines to support the patient care (cf. e.g. [KMH06]).

Despite the vision of a Semantic Web [BLHL01] and many efforts helping to realize it in the diverse application fields described above, the current Semantic Web still lacks of enough semantic data. Most information is still modeled and stored in relational databases and thus out of reach for many Semantic Web applications. Hence, the relational data has to be converted in a manual, error-prone, and time-consuming process or be mapped to the semantic models using system-specific mapping applications.

As a consequence, a technology to map relational data to the Semantic Web is required, which enables the user to create arbitrary mappings to Semantic Web ontologies, without having to adopt a novel mapping language. Additionally, this mapping technique should take the characteristics of relational databases into account, particularly data and schema evolution, i.e. enable users to access the databases transparently using Semantic Web techniques, without having to repeat the data translation process every time a modification occurs. Using such a technique, a user is rapidly able to provide Semantic Web applications with semantic rich data, which is actually stored in a relational database.

1.1 Contributions

Mainly situated in the gap between the relational and the semantic representation of data, this thesis comprises several relevant contributions for realizing the Semantic Web. To give the reader a synopsis of these contributions, they are summarized in this section.

- The main contribution of this thesis is the presentation of a two-tiered mapping technique from relational data to the Semantic Web. With the Relational.OWL representation of a relational database, we have introduced the only data and schema extraction technique, which automatically transforms the schema of any relational database into an ontology. The data stored in that specific database is then represented as an instance of this ontology. For most Semantic Web applications, such a representation is enough, since their aim is not to perform advanced reasoning tasks with this data, but to have a simple access. Nevertheless, we show how to extend our technique, mapping the Relational.OWL representation of the database to a target ontology with any Semantic Web query language, as long as it is closed within RDF.

- A further contribution of this thesis is a detailed analysis of the RDF query languages RDQL and SPARQL with respect to their expressiveness. Having no formal foundations, these languages are compared pragmatically with the main operations of the relational algebra. During our analysis we observed, that the main problem of RDQL is the absence of closeness, i.e. the result set of a query is not a valid RDF expression any more. SPARQL, the successor of RDQL, although being closed within RDF still has some drawbacks concerning its expressiveness. Nevertheless, SPARQL is already a big step towards the standardization of an RDF query language.

- Whenever relational data is mapped to Semantic Web objects, the information concerning the location in the original database gets lost. Nevertheless, it is essential to backtrack a resource to its original location for identification

matters, whenever reasoning tasks have to be performed. In this thesis we hence suggest the novel URI scheme `db` for identifying not only databases, but also their schema and data components like tables or columns. It is the only URI scheme, which combines the flexibility of the Semantic Web with the advantages of a global unique identifer and as far as we are aware, besides [BL98a] the only approach which addresses the problem of a global URI for databases at all.

- A further contribution of this thesis is the introduction of the Link Pattern Catalog as a modeling guideline for recurring problems appearing during the design or description of novel information and knowledge platforms. The Link Pattern Catalog consists of prototypes or solutions for recurring problems and therewith supports developers to model, describe, and understand complex information sharing environments like the Semantic Web. Furthermore the Link Patterns provide a common vocabulary for design and communication purposes, enabling developers to exchange successfully implemented solutions. For this purpose we additionally introduce the Data Link Modeling Language, a language for describing and modeling virtually any kind of data flows in information and knowledge sharing environments.

1.2 Outline

The outline of this thesis is as follows. In the next chapter we survey the background required for this thesis with a short introduction to the relational data model and the Semantic Web.

In Chapter 3 we present our technique to map relational data to the Semantic Web, which consists of two main steps. First, the Relational.OWL representation of a relational database is introduced, on which a consequent mapping to a specific target ontology can be created using any closed RDF query language.

Chapter 4 introduces the novel URI scheme `db` for relational databases, which is especially suitable for identifying the exact location of a data item in a relational database, but may be extended to support database of any data model.

The applications presented in Chapter 5 build up on the techniques described in the previous chapters. First, we introduce two different applications, which extract the data and schema information of a relational database and converts it to its Relational.OWL representation. Thereafter, we introduce RDQuery, a wrapper system which enables Semantic Web applications to access and query data actually stored in relational databases using their own built-in functionality, translating SPARQL and RDQL queries into SQL. The last application presented in this chapter is DÍGAME, an architecture for a P2P database, which achieves a reasonable tradeoff between autonomy and information sharing among both,

permanently available and volatile data sources using a Relational.OWL-based data representation.

Finally, Chapter 6 introduces the Link Pattern Catalog as a modeling guideline for recurring problems appearing during the design or description of novel information and knowledge platforms. Chapter 7 concludes this thesis.

Chapter 2

Background

Relational databases and the Semantic Web, in particular RDF as its main data representation format, are two different approaches for modeling and storing data permanently. Developed for the Web, RDF aims to collect information located on different sources for performing processing and reasoning tasks. Relational databases in contrast are created usually for specific tasks or applications, hence data already modeled in a remote node is often modeled again in the local database, resulting in a huge amount of data redundancy.

In this chapter we provide some background on relational databases and the Semantic Web. We first introduce relational databases (Section 2.1) and their data model (Section 2.1.1.1). Thereupon we present two different ways to access data stored in such databases, using the relational algebra with its basic operations (Section 2.1.1.2) and the Structured Query Language SQL in Section 2.1.2. Instead of giving a complete introduction to SQL, we present some of its advanced techniques which make SQL the unrivaled leader of query languages.

After describing the relational way to represent data, we introduce the Semantic Web (Section 2.2) and some of its techniques (Section 2.2.2), including RDF, RDF Schema, the Web Ontology Language OWL, and its query languages RDQL and SPARQL.

2.1 Relational Databases

With his initial work on the relational model [Cod70] and several subsequent publications [Cod72, Cod79], Edgar F. Codd revolutionized data storage and access fundamentally. Until then, storing data in database systems always meant to adapt the data to either the hierarchical or the network based models (cf. [Bac69, TF76, Bla98]). With the relational model, the database designer was no longer forced to adapt the structure of his data to these models, but was able to design the database more freely. A further disadvantage of those early database systems was the lack of a strict separation between the logical database model

and its actual physical implementation, resulting in several difficulties querying the data, ordering the results, or optimizing data access using index structures. The presentation of the relational model was a big step forward to eliminate these disadvantages (cf. [Cod70, GMUW02]).

Nevertheless, the relational data model would certainly not have reached its current popularity without a powerful query language. First introduced as SEQUEL [CB74], SQL soon became the de-facto standard for querying and manipulating relational databases in a structured way. It is remarkable, that the initial version of SEQUEL already supported the aggregation of data, grouping of data, or nested queries. These operations are often not even implemented in modern query languages (cf. Section 3.2). Recently the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) have published jointly the 2003 version of SQL [Int03b], which is divided in several parts (1-4, 9-11, 13, and 14) with a total of more than 3000 pages. SQL now provides besides the well-known query, update and data definition functionality, the possibility to include Java methods, XML, or external data in an SQL query.

Even modern database models like the object-oriented [SM96] or XML [SW00] did not succeed in replacing the relational model, but remain a niche technology virtually absorbed by modern relational databases (e.g. [STZ+99]).

Nowadays, the relational model is realized in numerous commercial and non-commercial database management systems and is recognized to be technically mature, hence frequently used exclusively because of its sophisticated query performance (e.g. [KCPA01], [PH03]). Currently there is an uncountable amount of relational database instances implemented around the world, ranging from small system specific databases to huge data banks with terabytes of data (cf. [Han98]).

2.1.1 Relational Model

In this section we give a short introduction to the relational data model and to its corresponding algebra. The introduction is based on [Cod70, Cha76, Cod79, LL95, SKS98, NE01].

2.1.1.1 Relational Data Model

Based on the mathematical concept of relations, the relational data model regards a database as a set of relations or tables. In turn, each relation R consists of a collection of attributes $A_1, A_2, ...A_n$, also stated as columns. Consequently, a relation R of degree n is a subset of the Cartesian product $A_1 \times A_2 \times ... \times A_n$ with the following properties:

- Each row is a distinct n-tuple element of R.
- Each value in a tuple is atomic, i.e. it is neither composite nor multivalued.

- The order of attributes and rows are irrelevant. Please note, that the work of Codd is inconsistent in this point, since he claims in [Cod70] the order of the attributes to be important, and in [Cod79] he specifies the order to be insignificant. Following the implementation in current relational databases, we hence regard the order of the attributes to be irrelevant.
- Both, relations and columns are labeled by names.

In Figure 2.1, a sample relation with four attributes and three rows is given. The relation is called *Address* and contains the attribute names ID, $Street$, ZIP, and $City$. Stated in the commonly used notation, our relation is $Address(ID, Street, ZIP, City)$ and contains as its values tuples like $t =< 2, UnterdenLinden, 10117, Berlin >$.

Address	*ID*	*Street*	*ZIP*	*City*
	1	Universitätsstr.	40225	Düsseldorf
	2	Unter den Linden	10117	Berlin
	3	Königsallee	40212	Düsseldorf

Figure 2.1: A relation with four attributes and three rows

2.1.1.2 Relational Algebra

The relational algebra is a query language for relational databases, i.e. a language in which a user can request specific information from a database. The result of such a request is again a relation, originated from one or more tables stored in the queried database, i.e. the relational algebra is a closed language. Since the result of querying relations is again a relation, we can produce chains of relational algebra operations, forming a relational algebra expression (cf. [NE01]).

The operations used to query a relational database defined within the relational algebra can be divided into three groups. The first group consists of the basic or fundamental operations, the second group of the composed operations, which can be stated in terms of the basic operations, and finally the third group of additional operations which are not based on the fundamental operations.

In this section, we give a short overview of the basic relational algebra operations and refer to [Cod79, SKS98, NE01, LL95] for a more detailed introduction. This section is based on [LL95] and hence adopts its notation. With $\mathcal{R}_m$ we denote the set of all m-ary relations, i.e. a relation $R \in \mathcal{R}_m$ has m attributes, denoted by $A_R = \{A_{R_1}, ..., A_{R_m}\}$. An attribute sequence $(A_{R_{s_1}}, ..., A_{R_{s_n}})$ of R is a tuple with $A_{R_{s_i}} \in A_R$, $s_i \in \{1, ..., m\}$, $s_i \neq s_j$ with $i \neq j$. Finally, the concatenation of the tuples $q = (q_1, ...q_n) \in Q$ and $r = (r_1, ...r_n) \in R$ of two relations $Q \in \mathcal{R}_n$ and $R \in \mathcal{R}_m$ is labeled with $q \circ r = (q_1, ..., q_n, r_1, ..., r_m)$.

Selection: The selection operation σ is defined as a function $\mathcal{R}_m \rightarrow \mathcal{R}_m$, which restricts the resulting relation to contain only those tuples $R \in \mathcal{R}_m$, which satisfy a given constraint $\Theta : R \rightarrow \{true, false\}$. A selection hence is defined as being $\sigma_\Theta(R) := \{r \in R \mid \Theta(r)\}$. Applying it to the example in Figure 2.1, a selection could be to filter the tuples to those having "Berlin" as the value of their *City* attribute, i.e. $\sigma_{City='Berlin'}(Address)$. The result of this operation is given in Figure 2.2. Following the definition of the θ-Select in [Cod79], the selection can contain the operators $<, \leq, \geq, =, >, \neq$, depending on the applicability to the corresponding attributes.

ID	*Street*	*ZIP*	*City*
2	Unter den Linden	10117	Berlin

Figure 2.2: Result of the $\sigma_{City='Berlin'}(Address)$ operation

Projection: To project ($\pi_{<attribute\ list>}(R)$) means to drop all the attributes from the relation R, which are not specified in the $<$ *attribute list* $>$, i.e. π is a $\mathcal{R}_m \rightarrow \mathcal{R}_n$ function with $(n \leq m)$ and a list of attributes $X = (A_{R_{s_1}}, ..., A_{R_{s_n}}) \in \mathcal{A}_R$, defined as $\pi_X(R) := \{(r_{s_1}, ..., r_{s_n}) \mid (r_1, ..., r_m) \in R\}$. Applying this to our example in Figure 2.1, the operation $\pi_{City}(Address)$ results in a relation containing the following two tuples: *(Düsseldorf, Berlin)*. The additional *Düsseldorf* entry was removed, since all duplicate tuples have to be eliminated.

Set Union: Unlike the selection and projection operations, the set union $\cup$ has to be operated on two union-compatible relations (cf. [Cod79]), i.e. it is a function $\mathcal{R}_m \times \mathcal{R}_m \rightarrow \mathcal{R}_m$, which collects all the tuples from the specified relations in a new one. Given $Q, R \in \mathcal{R}_m$ and equal attribute sets for Q and R, the set union is defined as $Q \cup R := \{q | q \in Q \vee q \in R\}$. Since the resulting relation is again regarded as a set, tuples contained in both relations are included only once.

Set Difference: Similar to the set union, the set difference $-$ requires two union-compatible relations as input, i.e. it is a $\mathcal{R}_m \times \mathcal{R}_m \rightarrow \mathcal{R}_m$ function. Given two relations $Q, R \in \mathcal{R}_m$ with identical attribute sets, the set difference is defined as $Q - R := \{q \mid q \in Q \wedge q \notin R\}$. This means, that the resulting relation contains all tuples from Q, which are not part of R.

Cartesian Product: The Cartesian product $\times$ of two relations unifies them into a new relation, containing the complete set of attributes from the two original relations, i.e. it is a function $\mathcal{R}_m \times \mathcal{R}_n \rightarrow \mathcal{R}_{m+n}$. Given two relations $Q \in \mathcal{R}_m$, $R \in \mathcal{R}_n$ with a disjoint set of attributes, the Cartesian product is defined as $Q \times R := \{q \circ r \mid q \in Q, r \in R\}$. The values of the resulting

relation are hence a combination of all tuples of the first relation with all tuples of the second relation. Do the original relations contain attributes with the same name, they are first renamed, adding an additional prefix to denote the name of the relation, this attribute came from.

2.1.2 SQL

In this section we point out the complexity, which the Structured Query Language (SQL) has gained throughout the years, from its beginnings as SEQUEL [CB74] to its current standardization as SQL 2003 [Int03b]. We hence do not give the usual kind of introduction to the basic functionality of SQL, but focus on the functionality, which makes SQL a sophisticated query language, especially comparing it with those discussed in Section 3.2. Since we only can describe a small subset of the huge amount of possibilities given by SQL, we refer to [SKS98, NE01, LL95, Mai83, Dat00]) and especially to [GMUW02], on which this overview is based, for a more detailed introduction to the Structured Query Language (SQL).

Data Manipulation: Contrary to the relational algebra, where data can only be queried, SQL enables the user to modify the data he queries either through inserting new data into the database, deleting some data, or modifying the data he queries in the relational database. The three keywords, with which a corresponding SQL instruction is initiated are `INSERT`, `DELETE`, and `UPDATE`. Most readers will most likely be familiar with this basic functionality of SQL, but taking the novel query languages for the Semantic Web into account, which in part do not support any data modification, this functionality becomes not as basic, as it seems at the first glance.

A tuple $(r_1, ...r_m)$ is inserted into a relation R with attributes $\{A_{R_1}, ..., A_{R_m}\}$ with the following SQL command

`INSERT INTO` $R(A_{R_1}, ..., A_{R_m})$ `VALUES` $(r_1, ...r_m)$`;`

It is not required to provide the attribute names, if all the data values are given in the required order, i.e. the order, the attributes were specified during the creation of the table.

To delete one or several tuples from a relation is very similar to selecting them, i.e. we use the same condition as if we would select them, but use the `DELETE FROM` keyword instead of a `SELECT`. Deleting some attributes from a relation R hence looks like

`DELETE FROM` R `WHERE <`*condition*`>;`

The last of the three data manipulation operands is used to modify data already stored in a relational database. Following the data deletion, we

again require a condition, specifying all the tuples of a relation, which shall be modified. Additionally, we have to provide the new values for the attributes to change. This is done using the `SET` keyword. Updating the tuples of a relation R matching the condition $< condition >$ with the new values specified in $< assign - new - values >$, hence looks like

`UPDATE` R `SET <`$assign - new - values$`> WHERE <`$condition$`>,`

whereas $< assign - new - values >$ is a chain of attribute/value pairs.

Subqueries: Partly introduced in the first version of SEQUEL [CB74], the possibility to use nested queries, enables the user to specify arbitrary complex queries. There are basically two different types of subqueries, depending on the place where they are stated: in the `FROM` or in the `WHERE` clause of the query.

If a nested query is specified in the `WHERE` clause of a query, the subquery is calculated dynamically and compared to the specified attribute. A typical example could be to retrieve all the persons from a salary database, which gain more money than the average employee, i.e.

```
SELECT *
FROM employee
WHERE salary > (SELECT AVG(salary)
                FROM employee);
```

Of course, such comparisons are only possible, if the inner query returns a single value. Otherwise, we would still be able to check, if the value of the attribute is included in the set of values returned by the inner query. This is done using the `IN` or `NOT IN` keywords instead of the usual relational operator, $>$ in our example.

If the result of a query shall not be compared with a single value of a relation, but be used as a virtual table, the subquery has to be specified in the `FROM` clause of the outer query. Since we have to assign the inner relation a virtual table name, its result can be referenced from anywhere in the outer query, e.g.

```
SELECT *
FROM employee, (SELECT ID, Street, City
                FROM Address
                WHERE City='Berlin') berl
WHERE salary > (SELECT AVG(salary)
                FROM employee)
  AND employee.addressid=berl.ID;
```

The sample query contains two subqueries and retrieves all the employees, who earn more than the average salary and live in Berlin.

Management of External Data: Since the first publication of the SQL/MED standard in SQL 2001 [Int01], users and applications are no longer restricted to access data stored in the local database system. Instead, introducing the `DATALINK` data type and the possibility to wrap remote systems, SQL/MED enables users to access or manage data stored in foreign (database) systems. The following short introduction is based on the 2000 version of SQL/MED described in [MMJ+01a]. For a more detailed specification of all the (current) possibilities provided by SQL/MED, we refer to [Int03d].

Throughout the years, database users already stored the path to external data files in their relational databases. This information was usually placed in a `VARCHAR` attribute, so applications could retrieve the information of the location of the file using SQL. A similar approach is followed by SQL/MED introducing the new data type `DATALINK`, which shall be used instead of the simple `VARCHAR` datatype to store the path to an external data file in the database. Depending on the actual implementation of the *datalinker*, i.e. the component responsible for the communication between the database and the local file system, using the `DATALINK` data type entails several advantages over the usage of a classical `VARCHAR` attribute, e.g. file recovery, access control management through tokens, or referential integrity checks, i.e. the corresponding value in the database is deleted whenever the local file is deleted or vice versa. The following example shows, how to create a table containing a `DATALINK` object without any referential integrity checks, i.e. the database system does not check, if the path to the file stored in that attribute, actually exists:

```
CREATE TABLE employee (
    ID INTEGER,
    name VARCHAR(255),
    picture DATALINK
            NO LINK CONTROL);
```

To provide the users and applications the possibility to query data stored in remote data sources, SQL/MED introduces a special wrapping mechanism. Being able to process only relational data, data in non-relational databases has to be represented as if it would be relational, to become queryable using SQL. As a result, we first have to instantiate a source specific *data wrapper* using a corresponding SQL statement on the local database:

```
CREATE FOREIGN DATA WRAPPER wrapper-name [...];
```

This wrapper is responsible for providing the relational view over the data source. After having created the wrapper, we have to create an instance of the remote server in our own database and link it with the corresponding *data wrapper*. This is done with the statement

```
CREATE SERVER server-name
   [...]
   FOREIGN DATA WRAPPER wrapper-name
   [....];
```

The last step is to provide access to the remote tables to local applications using the following statement

```
CREATE FOREIGN TABLE ft
   [ (col-def, coll-def,....)]
   SERVER server-name
   [ generic-options ];
```

After having successfully established such a linkage between the local database and the remote data source, the foreign table can be queried as if it would be a local one. In the following example, we use the remote table just created to join it with a local one:

```
SELECT *
  FROM ft, lt
  WHERE rt.ID=lt.ID;
```

SQL/XML: With SQL/XML [Int03c], the SQL standard leaves the relational world and enters the world of semi-structured data. The SQL/XML standard has three main topics, describing how to map relational data and schema components into XML and XML Schema, how to map SQL data types to XML data types, and how to use the newly introduced data type `XML` and its corresponding functions. Please note, that with SQL 2003, the SQL/XML data model was replaced by that of XQuery, introducing several subtypes for `XML` (cf. [EM04]). In this section we focus on the 2002 version of the standard, particularly on the `XML` data type, since it is more adequate for providing a first overview of the basic SQL/XML functionalities than the newer and at the same time more complex version of the standard.

The `XML` data type was introduced for enabling database users and applications to store, access, and query semi-structured data in relational databases. Such attributes may contain valid XML documents, forests of documents, text nodes, or mixed content (cf. [EM02]). Additionally, a database implementing the SQL/XML standard also provides the following functions to produce XML from the data stored in that database:

XMLELEMENT, XMLFOREST, XMLGEN, XMLCONCAT, and XMLAGG. In this section we present only some of these functions, a more detailed introduction is provided in [EM02, Tür03].

The XMLELEMENT function creates XML elements and allows two different arguments to be specified: the name and the content of the element to be created. The second argument, specifying the content of the element, may contain further, nested elements, text, or XML attributes using the XMLATTRIBUTES function, Based on the sample database from Figure 2.1, we are thus able to specify the following SQL query:

```
SELECT XMLELEMENT( NAME "Address",
                   XMLATTRIBUTES (a.id),
                   XMLELEMENT( NAME "ZIP",
                               a.ZIP)) AS output
  FROM Address a;
```

After processing the query, the database returns a relation which contains the corresponding XML values as its tuples:

output
<Address ID="1"><ZIP>40225</ZIP></Address>
<Address ID="2"><ZIP>10117</ZIP></Address>
<Address ID="3"><ZIP>40212</ZIP></Address>

The XMLGEN function creates, similar to XQuery and XSLT, XML documents from an XML skeleton, binding free variables to the responses of the query. The result of the following SQL query is hence the same as the result of the XMLELEMENT query specified above:

```
SELECT XMLGEN('<Address ID="{$ID}">
                        <ZIP>"{$ZIP}"</ZIP>
               </Address>',
               a.ID,
               a.ZIP) AS output
  FROM Address a;
```

SQL/XML provides three more functions for the translation of relational data into XML. The XMLFOREST function, which returns a forest of XML elements, the XMLCONCAT function, which also produces a forest of elements, concatenating the elements supplied as arguments, and finally the XMLAGG function, which creates a forest of XML elements aggregating XML values.

2.2 The Semantic Web

After having introduced the relational data model, and its query languages SQL and the relational algebra, we now focus on the Semantic Web. First, we explain the basic idea of the Semantic Web in Section 2.2.1. Thereupon, we go into more details, presenting RDF and OWL as the most relevant Semantic Web technologies, followed by a small introduction to some of their query languages in Section 2.2.2.

2.2.1 Introduction

With billions of static Web pages, the World Wide Web (WWW) in its current state is probably the largest information repository of the world. To this, we have to add the data of the deep or hidden web [Ber00], i.e. information usually stored in relational databases, made accessible through online forms or dynamic web pages. As a result, finding the right pages becomes a challenging task, which has to be mastered by anybody searching for specific information on the Web. Although search engine algorithms become better and better, the user still faces a massive information overload, where he has to decide on his own, which information is relevant and which is not. A search engine is only able to assist the user during this time-consuming process, since most of data stored on web servers, although being machine readable, is not machine understandable.

Realizing this situation, several approaches arose to automatically extract relevant information from web pages and thus help machines to understand them. Nevertheless, such knowledge extraction techniques (cf. [CDF+98, Fre98]) still provide faulty results, unacceptable in most scenarios with autonomously acting software agents applying inference rules and deducing additional knowledge, especially in life science scenarios, where the resulting information may be decisive for the therapy of a patient (cf. [KMH06]).

A different approach is followed by the so-called *Semantic Web*. In this next generation Web, software agents are not only able to process the data on a web site, but can understand its content and automatically deduce new knowledge using inference rules. For this purpose, each web page containing (relevant) information has to be annotated with additional meta data for being processable and interpretable by the corresponding software agents. Since the additional meta data has to be added manually, the drawback of this approach is obvious, since the whole annotation process is time-consuming and error-prone.

Currently there are two competing approaches in trying to realize the Semantic Web: Topic Maps and RDF, promoted by the International Organization for Standardization (ISO) and the World Wide Web Consortium (W3C) respectively. The first approach models knowledge using Topics as their central concept, enriched with Topic Names, Topic Occurrences, Associations, etc. More information concerning Topic Maps can be found in [Int03a, WM02, Pep00].

A less complex data model was introduced by the W3C with the Resource Description Framework (RDF). Having only three different concepts (Resources, Properties, and Literals) and statements, which are easy to create and understand, this Semantic Web language allows to model virtually anything related to anything. Director of the W3C and named to be the inventor of the World Wide Web, Tim Berners-Lee was one of the first persons to publish his idea of a Semantic Web [BL98b, BLHL01, BLM02]. Following his idea of software agents deducing knowledge from information stored on the Web, the W3C started with RDF [MM04] an initiative to realize this idea.

Being stated as the next generation Web, the idea of the Semantic Web is often misunderstood to be restricted to sharing information over the current Web. Instead, these techniques can be applied to any application field, where knowledge is shared and software agents shall perform reasoning tasks using inference rules. Certainly the Semantic Web enables personal digital agents, as described by Tim Berners-Lee et al. [BLHL01] to collect and evaluate relevant information over the Web, but there are also several application fields not directly connected to the current Web, including knowledge management, data integration [Av04, SL03], or drug discovery tasks [SMQ06, Neu05].

2.2.2 Semantic Web Technologies

In this section we focus on the Semantic Web strategy followed by the World Wide Web Consortium and its Director Tim Berners-Lee. For that purpose, we describe the Resource Description Framework (RDF) and the Web Ontology Language (OWL) as its core technologies. Additionally, we present sample RDF query languages and describe how to use them to access information modeled and stored using RDF.

2.2.2.1 RDF

The Resource Description Framework (RDF) [MM04] is the core technology recently recommended by the W3C for realizing the idea of a Semantic Web. The main advantage of RDF is its simple data model, which is absolutely domain-independent and easy to understand by both, humans and machines. The user is not restricted to specific concepts for modeling his view of a world, but may create his own classes and instances, similar to how it is done in object-oriented programming languages. Nevertheless, if each user models his own view of the world, an automatic reasoning task performed by software agents becomes challenging. An agent may hardly identify a concept *bus* from one user to be identical to a concept *autobus* from another user. Hence, it is advisable to use a commonly available data model, or at least provide a mapping to existing models, for enabling software agents to realize such reasoning tasks. The following introduction

to RDF is based on [Av04, MM04, Be004, BGM04], to which we refer for a more detailed description.

RDF Model:

The main building blocks of RDF are *statements*, *resources*, and *properties*. A statement is a triple consisting of a *subject*, a *predicate*, and an *object*, similar to a sentence expressed in the English language (cf. [Av04]). The subject, which is actually being described in the statement, is a resource, i.e. a thing. The property, corresponding to the verb of the sentence is always a predicate, whereas the object of the statement is again usually a resource. In order to identify a single object or property within the Semantic Web, each concept, but anonymous resources or blank nodes, has assigned an Uniform Resource Identifier (URI) (cf. Section 4). Since it is difficult to include a string representation into such a model, RDF introduces a further concept: the *literal*. It represents a string value and may be used instead of a resource as the object of a statement. It is not allowed to be used as a subject.

After having introduced the main concepts of RDF, we can express the following sentence in terms of RDF:

> The creator of this thesis is Cristian Pérez de Laborda.

Choosing *thesis* to be the subject of our statement, *has creator* becomes the predicate, and *Cristian Pérez de Laborda* the object of the RDF correspondent. For identifying the resources we are describing and for being able to refer them from external documents, we introduce additional URIs. The URI of this thesis could be e.g. `http://diss.de/ebib/diss/file?dissid=1041` and for the property e.g. `http://purl.org/dc/elements/1.1/creator`. The creator of the thesis is only known by its name, i.e. we do not have an URI for his identification. If we split the full name of the author into its components and represent them using literals, the sentence given above results in an RDF graph as depicted in Figure 2.3. The corresponding RDF graph contains three adjacent statements, one about the thesis itself and two about the blank node representing the author of the thesis.

Besides the concepts presented above, RDF introduces some advanced constructs to allow the creation of more complex relationships between resources. A *bag* for example is a resource, which contains an unordered set of resources or literals. In contrast, the *sequence* contains an ordered list of resources and literals. For a complete list of possible concepts and a more detailed description of RDF, we again refer to [MM04, Av04].

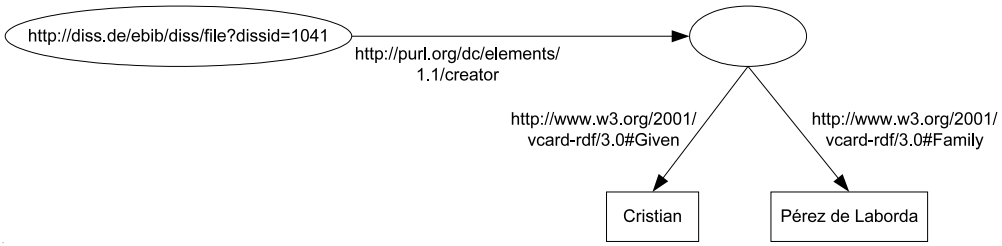

Figure 2.3: A sample RDF graph

RDF Schema:

As already mentioned, RDF does not specify or impose any rules to the domain of a given RDF statement. Hence, the user himself is responsible to use the appropriated vocabulary required for his specific domain. This vocabulary has to be specified using RDF Schema (RDFS) [BGM04].

Despite being able to relate two properties using an RDF predicate, we cannot state anything about its semantics, i.e. the meaning of this statement. Following the example of the object-oriented software design, RDF Schema introduces the concepts of *classes* (`rdfs:Class`) and *properties* (`rdfs:Property`). The class concept is used to define types of objects, hence a resource as introduced above is an instance of an RDFS class. Furthermore, RDFS allows the creation of new properties, which may then be used as predicates in RDF triples. Similar to the creation of a class using class-inheritance, existing and new properties can be arranged using sub-property relationships.

RDFS does not introduce a novel data model, but uses RDF to state its assertions, i.e. each RDFS statement is also a valid RDF graph. For instance, the property `http://www.w3.org/2001/vcard-rdf/3.0#Given` used in Figure 2.3 is defined with RDFS using an RDF graph as presented in Figure 2.4. Since RDFS definitions are always RDF, we are able to use both, RDF and RDFS concepts for specifying the properties and classes to be created.

If the `domain` or the `range` of a property shall be restricted to a specific set of resources, this can be done using corresponding properties. A detailed description of how to use these properties is given in [BGM04].

RDF Syntax:

We have described the data model of RDF in the previous sections and have explained how to extend it using RDFS. Although we already used a graph-based representation for our examples, we did not yet introduce a concrete syntax representation for RDF. Like many modeling languages,

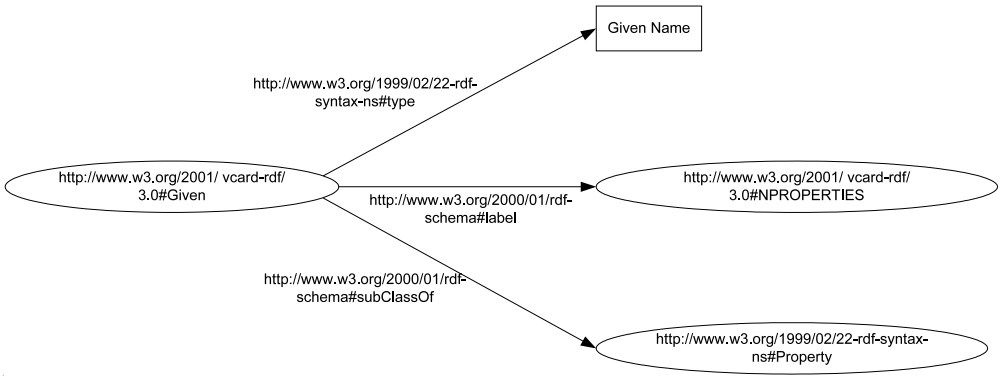

Figure 2.4: A sample RDFS graph

RDF is not restricted to a specific syntax, but provides several different ways to represent the information modeled.

Probably the most intuitive way to represent RDF statements is using a graph-based diagram as shown in Figure 2.3 and Figure 2.4. Despite being easy to understand by a human, this representation is rather difficult to be implemented in a machine-processable and exchangeable way. Hence, such a graphical representation is only suitable for providing humans an insight to the information (to be) modeled and not for autonomously interacting software agents.

A further straightforward way to represent the triples of an RDF graph is to describe them as they are, i.e. as triples consisting of URIs, blank nodes, or literals. Since a blank node does not contain an URI and it could be described with more than one property, we have to assign a temporary name, for being able to reference the blank node from the corresponding triples. This temporary URI is only valid for the purpose of referencing it within the same collection of triples (cf. [KlC04]). Taking the graph of Figure 2.3 as an example, the corresponding triple representation including the blank node for the creator of the thesis are described as presented in Figure 2.5.

```
http://diss.de/ebib/
      diss/file?dissid=1041   http://purl.org/dc/elements/1.1/creator       _:owner.
_:owner                       http://www.w3.org/2001/vcard-rdf/3.0#Given    "Cristian".
_:owner                       http://www.w3.org/2001/vcard-rdf/3.0#Family   "Pérez de Laborda".
```

Figure 2.5: Triple representation of the sample RDF graph

To represent an RDF graph using triples has several drawbacks, since this form od representation is rather inflexible lacking of metadata, i.e. no char-

acter encoding is given, the position of a triple element is not changeable, and the delimiter between the elements of the triple are defined to be a blank. This information is either hard-coded into the representation format or not provided at all. Consequently and although being processable by a machine, the implementation of this representation is a challenging task and hence not advisable.

The most popular way to represent an RDF data model is to use an XML-based syntax. Specified in its own W3C Recommendation [Be004], the RDF/XML Syntax represents the RDF graph in terms of XML. To describe the sample RDF graph of Figure 2.3 using RDF/XML results in an XML document as given in Figure 2.6.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#">
  <rdf:Description
       rdf:about="http://diss.de/ebib/diss/file?dissid=1041">
    <dc:creator>
      <rdf:Description>
        <vcard:Given>Cristian</vcard:Given>
        <vcard:Family>Pérez de Laborda</vcard:Family>
      </rdf:Description>
    </dc:creator>
  </rdf:Description>
</rdf:RDF>
```

Figure 2.6: RDF/XML representation of the sample RDF graph

In the XML representation of our sample RDF graph, the corresponding XML version and encoding are specified first. Then, an `rdf:RDF` tag with the required namespace definitions (cf. [BPSM04]) is provided, signalizing the beginning of the RDF document. Contrary to ordinary XML documents, where namespaces are only used to prevent ambiguity in the label of the elements, in RDF they refer to external resource definitions, i.e. RDFS documents. Agreeing on the concepts defined in such files, other users may also reference to these RDFS documents from their RDF graphs. Such a common representation of abstract objects is the first step in creating a large, distributed, and common knowledge representation (cf. [Av04]).

After initiating the RDF document, an `rdf:Description` element is opened representing the first subject of our RDF graph. Giving its URI in the `rdf:about` attribute, we unambiguously specify the subject we are describing to be the specific thesis. The `rdf:about` attribute can be replaced by `rdf:ID`, if the complete URI of the current document is part of the URI of

the element, i.e. giving a relative URI with respect to the current document (cf. [Be004]).

The space between the opening and the ending tags of the `rdf:Description` element are used for the detailed specification of the thesis. the only property of the thesis object is represented using a `dc:creator` element. Its value is then represented by a blank node, i.e. an `rdf:Description` element without an URI. The last two elements defined in the RDF document are the `vcard:Given` and `vcard:Family` elements of the blank node with their corresponding values. Both attributes are in turn nested within the blank node representation.

Having given only a small overview of the possibilities provided by RDF/XML to represent an RDF graph in a machine processable way, we again refer to [Be004] for a complete introduction to the RDF/XML syntax.

2.2.2.2 OWL Web Ontology Language

Because of the expressiveness of RDF and RDF Schema, which are basically limited to simple object, class, or property relationships, the W3C realized the requirement for a modeling language going beyond the basic semantics of RDF Schema. Lacking enough support for shared ontology definitions, ontology evolution, interoperability, inconsistency detection, or expressiveness, the W3C decided to adapt the basic ideas implemented in DAML+OIL [MFHS02] into an own web ontology language called OWL (cf. [BvH+04, Hef04, Av04]).

Borrowed from philosophy, where an ontology is a systematic account of existence, in knowledge-based information systems, this term means the explicit specification of a conceptualization, i.e. a computer processable representation of things that exist and their relationships among each other [Gru93, Hef04]. Contrary to taxonomies, where these relationships are restricted to build up hierarchies, ontologies allow the creation of graphs with arbitrary relationships among its components.

The OWL Web Ontology Language [Mv04] builds up on the RDF layer of the Semantic Web and describes similar to RDFS formally the meaning of terminology used in RDF documents, adding additional vocabulary to RDF Schema, e.g. a disjointness relation between classes, transitivity and cardinality of properties, or the creation of complex classes. Since applying a full logic to the expressiveness of RDF leads to a certain undecidability of some problems in the resulting language, the W3C has split OWL into three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full. The following introduction to the three sublanguages is based on [BvH+04, Hef04, Av04], which should be considered for a more detailed description.

OWL Lite is the most basic of the three OWL sublanguages, supporting classification hierarchies and simple constraints. Additionally, OWL Lite pro-

vides the possibility to build up subclass hierarchies and make properties optional, i.e. to impose a cardinality constraint of 0 and 1.

OWL DL adds additional features to OWL Lite, corresponding to the expressiveness provided by description logics (cf. [BCM+03]). Unlike OWL Lite, OWL DL supports all language constructs of OWL, but still imposes some restrictions, e.g. an instance of a class cannot be at the same time itself a class or a property. These restrictions are required to guarantee the computational completeness and decidability of OWL DL.

OWL Full is the OWL sublanguage, which supports the complete functionality of OWL and allows an arbitrary combination with any constructs of RDF or RDF Schema. As a result, the usage of OWL Full may result in undecidable problems. In fact, even the authors of the OWL recommendation [Mv04] disbelieve that reasoning software agents will be able to support the full functionality of OWL Full. OWL Full is the only of the three sublanguages, which is completely compatible to RDF, i.e. any legal RDF document is also a legal OWL document and vice versa. For OWL Lite and OWL DL, only holds, that any legal OWL document is a legal RDF document.

2.2.2.3 RDF Query Languages

Being able to model information using RDF and its derivates, Semantic Web applications need a standardized access to such data. Taking into account huge data models like *Wikipedia*[3] [Wik06] with more than 47 million triples, it is obvious, that a simple access to such data banks is not enough and hence a suitable query language is required.

Indeed, RDF can be queried using any XML query language like XQuery [BCF+05] or a transformation language like XSLT [XSL99], since we are able to provide an XML representation of the RDF model (cf. Section 2.2.2.1). Nevertheless, such approaches ignore the enhanced expressiveness of RDF compared to XML, like its graph-based structure, class-inheritance, triples, or inference rules.

Although there are many different query languages for RDF, ranging from logic-based languages like TRIPLE [SD02] to XQuery or XSLT-based query languages like RDF Twig [Wal03], in this section we focus on RDQL [Sea04] and SPARQL [PS06b] as RDF query language representatives. For a quite complete overview of current RDF query langues we refer to [BBFS05, HBEV04], on which we partly base this introduction.

RDQL: was submitted to the W3C as a member submission (cf. [Jac05]) in 2004 by Andy Seaborne from the HP Labs in Bristol [Sea04]. It was soon implemented in the Jena Semantic Web Framework [Jen06] for Java and hence became rapidly popular within the Java community.

An RDQL query consists of several graph patterns, expressed in triples containing named variables, RDF values, and additional restrictions to the values of these variables. The results of such a query are possible variable bindings with respect to the original RDF graph, matching both, the triple patterns and the additional constraints. RDQL queries remind of SQL and are composed as described in Table 2.1.

`SELECT`	list of free variables to be bound and returned
`FROM`	URIs specifying the RDF models to be queried
`WHERE`	triple patterns including variables
`AND`	additional constraints for the values of the variables
`USING`	namespace definitions

Table 2.1: RDQL clauses

If we want to use RDQL to retrieve the last name of the author described in the RDF graph of Figure 2.3, we can use the following query:

```
SELECT  ?d
WHERE   (?a, dc:creator, ?c),
        (?c, vcard:Family, ?d)
AND     ?a EQ <http://diss.de/ebib/diss/file?dissid=1041>
USING   vcard for <http://www.w3.org/2001/vcard-rdf/3.0#>,
        dc    for <http://purl.org/dc/elements/1.1/>
```

In the first triple of the `WHERE` clause, first a variable `?a` is introduced. The object represented by this variable must have a property `dc:creator` with a value represented by the variable `?c`. Additionally, the object represented by `?c` must have a `vcard:Family` attribute with a value represented by `?d`. Since the variable `?a` is bound to the URI of our concrete thesis in the `AND` clause of the query and the thesis has only one creator, the object represented by `?c` is the blank node assigned to the `dc:creator` property of the thesis. As a result, the `?d` variable contains the last name of the author of the thesis, which is the value returned to the user, since it is the only variable specified in the `SELECT` clause of the query. The namespaces for the `vcard` and `dc` ontologies are given in the `USING` part of the query. Since no `FROM` clause is given, the query is applied to the model known from the context. The query returns `"Pérez de Laborda"`, as expected.

SPARQL: Realizing the need for a standardized query language and the drawbacks of the member submission RDQL, like lack of expressiveness or closeness (cf. Section 3.2), the W3C decided to promote an own RDF query language called SPARQL [PS06b] based on RDQL and its successor BRQL [PS06a]. Although still a candidate recommendation (cf. [Jac05]), SPARQL

is already supported (in part) by many applications (e.g. Jena [Jen06]) and some major database vendors (e.g. Oracle [CDES05]) and will hence most likely become the de facto standard for querying RDF graphs.

Based on RDQL, a SPARQL query also consists of several graph patterns expressed with triples in an SQL-like syntax. It extends RDQL adding for example the possibility to express closed RDF queries, either constructing a new RDF graph or returning a subgraph of the RDF model queried. A SPARQL query is composed using the main clauses described in Table 2.2.

`PREFIX`	for defining the namespaces
`SELECT (DISTINCT)`	list of free variables to be returned
`CONSTRUCT`	to create a new RDF graph, used instead of `SELECT`
`DESCRIBE`	retrieve a complete resource, used instead of `SELECT`
`ASK`	returns `yes` or `no` for possible graph patterns
`FROM`	URIs specifying the RDF models to be queried
`WHERE`	triple patterns including variables
`ORDER BY`	applies ordering conditions to a `SELECT` query
`OPTIONAL`	indicates that a graph pattern is optional
`FILTER`	subset of XPath filters, applied to variables

Table 2.2: Main SPARQL clauses

Following the example given in the previous section, we now query the given RDF model from Figure 2.3 using SPARQL to retrieve the complete resource representation of the author. Instead of retrieving possible variable bindings, as done with RDQL, we now demand the result set to be a valid RDF document. This can be achieved with the following query:

```
PREFIX   dc: <http://purl.org/dc/elements/1.1/>
DESCRIBE ?c
WHERE    {{?a dc:creator ?c}.
       FILTER ?a=<http://diss.de/ebib/diss/file?dissid=1041>}
```

Unlike in RDQL, first the required namespace is introduced in the `PREFIX` clause, i.e. `dc` for the Dublin Core ontology. Thereupon, depending on the functionality required, either a `SELECT`, `CONSTRUCT`, or `DESCRIBE` clause is specified. Since we want the complete resource representing the author to be returned, we use the `DESCRIBE` keyword followed by the variable representing the resource to be retrieved. The required triple-constraints are then specified as usual in the `WHERE` clause. Instead of using an `AND` clause for adding additional constraints like in RDQL, SPARQL uses a `FILTER` subclause, included within the `WHERE` part of the query.

Chapter 3

Bridging the Semantic Gap

The vision of a next generation Web [BLHL01] is, despite all the efforts to build up the Semantic Web, still more dream than reality. The main reason for this issue is the lack of data, since the vast majority of information is still stored in (relational) databases and thus unavailable for most Semantic Web applications. As a consequence, local applications usually create their own manual relational to semantic mappings, accessing the relational data with SQL. Due to the fact, that such mappings have to be created manually, the consequences are obvious: different applications could map identical data extracted from the same database to different concepts of the Semantic Web. Consequently, a well-defined mapping of relational to semantic data is required.

In this chapter we first introduce the Relational.OWL representation of a relational database. This technique enables us to automatically transform the data and schema components of a relational database into a Semantic Web representation. Initially, the schema of the database is translated into a schema ontology, on which thereupon, the representation of the actual data can be based on. After having created an automatic transformation mechanism into a representation, processable by most Semantic Web application, we analyze if current query languages for RDF and OWL are expressive enough to replace together with Relational.OWL existing SQL access to relational databases. The advantage of such an approach is obvious, since Semantic Web applications could use their own built-in functionality to access data actually stored in relational databases.

Being processable by most Semantic Web applications, the Relational.OWL representation still has one fundamental drawback: the data is represented as it was stored in the relational database, i.e. within corresponding Table and Column objects. For most applications, such a representation should be enough, since their aim is not to perform advanced reasoning tasks with this data. Nevertheless, some applications still require the data to be represented in terms of a specific ontology. We hence show, how to extend our Relational.OWL technique to obtain a mapping from the Relational.OWL representation of a relational database to a

target ontology, using any Semantic Web query language, as long as it is closed within RDF.

The remainder of this chapter is organized as follows: In Section 3.1 we introduce Relational.OWL, our technique to represent the data and schema components of a relational database in a machine processable and understandable format. Section 3.2 analyzes, whether the RDF query languages RDQL and SPARQL could replace the existing SQL access to relational databases from Semantic Web applications. Since the Relational.OWL representation does not result in objects containing *real* semantics, we show in Section 3.3, how to create mappings from the relational model to a target ontology. Section 3.4 catches up related work and Section 3.5 concludes giving some hints for further research. The approaches presented in this chapter are partly based on work published in [PC05a], [PC05b], [PC06a], and [PC06b].

3.1 Relational.OWL

In this section we introduce *Relational.OWL*, an OWL Web Ontology Language-based (cf. Section 2.2.2.2) representation format for relational data and schema components, which is particularly appropriate for exchanging items among remote database systems. OWL, originally created for the Semantic Web, enables us to represent not only the data itself, but also its *interpretation*, i.e. knowledge about its format, its origin, or its original usage within specific frameworks.

With Relational.OWL, we are able to automatically extract the schema information of any relational database and regard it as a new ontology. Thereupon, the corresponding data items can be represented as instances of this data source specific ontology. Using this technique, potentially any relational database may automatically become an integral part of the Semantic Web. The advantage of such an approach for the Semantic Web is obvious, since it enables Semantic Web applications to perform reasoning tasks accessing or even querying data actually stored in relational databases using their own built-in functionality.

A further application area for Relational.OWL, which apparently has little in common with the Semantic Web, is the data and schema exchange among relational databases. Using RDF and OWL as a common representation language, remote databases are instantly able to understand each other without having to arrange an explicit exchange format — the usage of a common ontology is enough. This feature would be impossible using present XML formats. The broad application field of this proposal hence includes all types of (multi)database systems [LA86] where component databases share schema and data information, as done in Peer-to-Peer or Peer-to-Multi-Peer databases [HIMT03].

Contrary to present approaches where RDF is stored in relational databases, and relational data is converted into RDF [Mel01], this approach aims at bringing together the representation of both, legacy data and schema components with a

common mediated language based on RDF and OWL, the powerful Semantic Web languages recently recommended by the World Wide Web Consortium [MM04, Mv04]. In other words, we have created a technique to automatically transform the schema and data components of a relational database into a representation format based on the knowledge representation techniques of the Semantic Web [BKD+01].

Adopting the opportunities given by the Semantic Web and our novel Relational.OWL ontology, we are able to uniformly describe and share the schema and data items of virtually any (relational) database. Additionally, this data becomes processable by most Semantic Web application and hence an integral part of the Semantic Web.

The data and schema extraction process presented in this section can be divided into two major steps. First, the schema of the database is described using the Relational.OWL ontology. This schema representation itself is then used as a new ontology, on which the actual data representation is based. As a result we obtain a model with three layers with the Relational.OWL ontology on the bottom (Figure 3.1). The layer above stands for the schema ontology, whereas the concrete data representation is placed on the top-most layer. Since we want to represent all three layers using OWL, we need the entire language (i.e. *OWL Full*) and not one of its sublanguages *OWL Lite* or *OWL DL*.

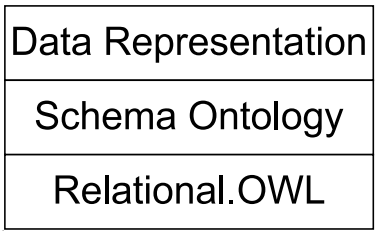

Figure 3.1: Three layers of abstraction using Relational.OWL

3.1.1 Motivation

In this section we present the main application fields of Relational.OWL, where a relational data and schema representation based on the Web Ontology Language can substantially improve the corresponding data sharing, accessing, or querying processes.

3.1.1.1 Data for the Semantic Web

Despite the vision of a Semantic Web [BLHL01] and many efforts helping to realize it, the current Semantic Web still lacks of enough semantic data. Most information is still modeled and stored in relational databases and thus out of

reach for most Semantic Web applications. As a consequence, such applications need to create a corresponding mapping between the relational and the semantic models by themselves for being able to access relational data. Realizing this situation, some efforts have arisen to straighten out this deplorable situation. Most of these approaches translate relational data into a Semantic Web representation using a proprietary mapping language (cf. Section 3.4).

With Relational.OWL we provide a technique to automatically transform relational data into a machine processable and understandable representation. As a consequence, potentially each relational database may become an integral part of the Semantic Web without any human intervention. The advantages of such an approach are obvious, since it enables Semantic Web applications to access and query data actually stored in relational databases using their own built-in functionality. Indeed, such a representation does not include *real* semantics, since it converts the schema of a database automatically into an ontology containing table and column objects and the data items as its instances, i.e. the data is described as it was in the database (cf. Section 3.3). Nevertheless, for many Semantic Web applications, particularly in scenarios with frequently changing or evolving schemas, this is a reasonable technique. Semantic Web applications are quickly able to access legacy data stored in a relational database using their own built-in functionality, without having to manually adjust the mapping each time the schema changes.

3.1.1.2 Relational.OWL as an Exchange Format

The representation of relational data and schema with the OWL Web Ontology Language entails several advantages over classical (semi)structured exchange formats like XML [BPSM04]. In this section we discuss these advantages and explain, why the usage of OWL should be considered, although a minor increase of data overhead has to be taken into account.

Knowledge Representation: The knowledge representation approach of OWL enables us to write formal conceptualizations of domain models, the so-called ontologies [Av03]. Having created such an ontology we are able to encode knowledge about things and their interrelationships within our specific domain into a machine–understandable format, which can afterwards be decoded and interpreted by any remote node or peer which has access to that ontology (cf. [BOF+04]).

Applied to the domain of relational databases, we can describe data and schema items and its corresponding interconnections in a machine–processable and understandable way, as soon as we have defined an ontology for the representation of relational data(bases).

Reliable Data and Schema Exchange: The only way to guarantee the faultless interpretation of data and schema items on a remote node is knowledge

representation. The knowledge representation process prevents machines on different sites to interpret the same data differently. Thus, it can be guaranteed that an item exchanged among remote peers will always maintain precisely its intended meaning.

The risk of a possible misunderstanding can usually be minimized through a face to face communication or previous agreement between the exchange partners. Nevertheless, this can not be accomplished within a volatile environment, where peers may appear and disappear at any time (e.g. Peer-to-Peer databases). In this case it is vital to have a representation format which is unambiguous for all exchange partners involved.

No Explicit Exchange Format: It is very sophisticated to arrange a common representation format for a data exchange, especially if the partners involved barely know each other or the schema changes constantly. In the latter situation, a communication channel set up by two nodes may probably be used only once, thus the arrangement of a proprietary format would be a tremendous overhead.

Although it is possible to arrange such formats (in)formally, this leads to unmanageable amounts of representation formats, particularly if a node is involved in several data exchanges. As we have discussed above, the usage of a (semi)structured format in its classical way could cause misunderstandings among the sites involved. Thus, using knowledge representation techniques enables remote peers to understand the information provided without having to arrange a specific exchange or representation format, since it is provided with RDF and OWL (e.g. the OWL XML representation [HEPS02]). The only requirements for establishing such a substantial communication are components capable to handle such knowledge representation formats and a common ontology like *Relational.OWL*, which is presented in this section.

Convertible Representation: One of the main advantages of using common knowledge representation techniques is the simple interconnectivity of existing ontologies. Two communities using different ontologies for the representation of relational databases could easily collaborate, as soon as a semantic mapping between these ontologies is created [DMDH02].

Having such a mediator ontology, both communities are instantly able to understand the representation format of each other, without having to change a single thing on their data and schema import or export processes. The interpretation of mediator ontologies is an integral part of the knowledge representation techniques used in RDF and OWL.

Data as an Instance of its Schema: Given the fact that OWL enables the creation of classes and its instances with one and the same syntax, we

are able to describe relational schema and data items. Furthermore we link schema and data representation in a singular way resulting in a homogenous data and schema format, where data items are defined as instances of their own schemata. This representation corresponds exactly to the internal representation used by current relational database systems.

Uniform Representation: Since we have to describe data and on a higher abstraction level its schema, we require two different representation formats, especially if we want to have the data represented as an instance of its schema. OWL supports both, ontological modeling and reasoning using the same syntax. We thus can provide an uniform framework for the precise representation of relational schema and its data items. Contrary to this, other languages like XML need to fall back to their corresponding modeling languages called XML Schema, or Document Type Definition (DTD).

Reasonable Data Overhead: Due to the characteristics mentioned above, especially concerning the powerful ontological meaning and reasoning implemented in OWL, we have designed a promising technique for the representation of relational data and schemata, which is more powerful than the established representation or exchange formats like genuine XML. In order to achieve this extended functionality we have to accept an increased amount of data. For a detailed analysis on the data overhead produced by Relational.OWL, we refer to Section 3.1.6.

Applying knowledge representation techniques to the field of data and schema extraction results in various advantages, which could have a big impact especially within Peer-to-Peer databases, where volatile peers may appear for a short time offering or demanding for data. As long as all partners understand the Web Ontology Language OWL, we do not need to negotiate a data or schema exchange format any more, since all partners involved are capable to talk the same *language* Relational.OWL.

3.1.1.3 Peer-to-Peer Databases

Having different aims to achieve, most Peer-to-Peer (P2P) databases (cf. [HIMT03, NWQ+02] or Section 5.3) are based on the same principle: data stored in one single peer has to be made accessible to other remote peers and vice versa. Afterwards this data can be requested, queried, replicated, or integrated depending on the purpose of the remote system. Nevertheless, sharing relational data in such environments is a challenging task, since there are several substantial differences from conventional filesharing systems.

Volatile Peers: Peers of a P2P network are usually autonomous. This autonomy includes the right to decide whether to join or to leave an information

sharing environment at any time. Such volatile peers may appear shortly, collect or deliver some data, and disappear again. It even can not be assured that a peer joins the network ever again. Due to the short availability of potentially any peer, the negotiation of an exchange protocol for data and schema items becomes quite challenging. An exchange format, which can be understood instantly by all exchange partners would be more useful.

Data Distribution: Due to the characteristics of P2P environments, data which is significant for a peer may be spread over numerous data sources. Thus, this peer is required to collect that information from several remote data sources. In order to be able to receive and understand such data, the exchange partners need to arrange a data and a schema representation format. At worst, a peer would have to interpret a different format for every single data flow causing an unmanageable situation.

Relational Data: In classical filesharing networks, where textual or binary data (e.g. music) is offered, a file contains all the information required for understanding this data. Whereas sharing relational data means to offer data enriched with vital schema information including important instructions on how to interpret and use that data correctly (e.g. table and column names, consistency constraints, etc.). This metadata has to be transferred separately as long as data and metadata shall not be mashed.

Data Evolution: Data distributed over classical filesharing networks does usually not change, i.e. a file containing a music song will be identical, no matter how much time has passed. Whereas it is most likely, that a relational database will evolve after a certain time period, resulting in changes concerning both, data and schema components. This fact has to be taken into account within the relational data sharing process.

As a result, sharing relational data within a Peer-to-Peer environment means to distribute not only data items themselves, but also their schemata among multiple previously unknown peers. We thus need an exchange format, which on the one hand can be understood by a broad community of peers without being explicitly arranged beforehand and which on the other hand has to be suitable for representing relational schemata and their corresponding data instances.

Relational.OWL is not a classical exchange format, but a representation technique, which is equally suitable for data backups or migration scenarios. Anyway, Relational.OWL itself may easily be extended to fit all requirements of a data exchange format including modification, addition, or deletion directives.

In fact, including this information into our data and schema representation would mean to loose the independence from the application fields, i.e. such an exchange format could not be used for enabling Semantic Web applications to access data stored in relational databases any more. We thus have decided not

to include exchange or replication specific instructions into Relational.OWL, but to keep it an application independent representation technique.

3.1.2 Relevant Metadata

A database management system maintains a huge amount of metadata information to manage the whole system in a proper way. In current relational database systems, this information is stored in predefined system-tables, also called *Data Dictionary* or *Repository*.

We have decided to include only the upmost relevant metadata into our Relational.OWL ontology, since a large amount of the data stored in the repository is very system-specific and thus not required or suitable for a semantic representation. The metadata we have selected is the required one for a proper interpretation of the actual data (not metadata, cf. Section 3.1.5) representation. In fact, the set of schema items described may easily be extended, if it is required later on. As a result, we have included the following concepts into the Relational.OWL ontology:

Tables and Columns: As implied by its name, the most important schema component of a relational database is the relation, also called table. Additionally each table consists of columns (attributes), where the actual data is stored. Both schema components are the upmost essential information for representing the schema of a relational database and thus have to be included in the Relational.OWL ontology.

Primary and Foreign Keys: One or more columns may compose the primary key of a table. This information can be very useful for a target system, in particular if data updates have to be synchronized. Otherwise problems could arise assigning the new values to the old ones. Foreign Keys have to be represented analogously, otherwise the data in a target system could become inconsistent.

Data Types: Data types restrict the possible values in a column (e.g. only `integer` or only `varchar` values). This special form of consistency constraints can be indispensable if a bidirectional data synchronization is being performed and very useful for performing a small consistency check of the data received by a potential target system.

OWL provides built-in datatypes and the possibility to fall back to the XML Schema datatypes [BM04]. Since there is no standard way to use the latter within OWL [PSH04], we have decided to restrict this first version of Relational.OWL to those datatypes clearly defined within the OWL abstract syntax. Nevertheless, we need a technique to represent possible restrictions concerning the maximal length of values stored in each column (e.g. `varchar(100)`).

Concluding, it is necessary to include a representation for

- tables,
- columns,
- datatypes possibly with length restrictions,
- primary keys,
- foreign keys, and
- the relations among each other

into the Relational.OWL ontology.

3.1.3 The Relational.OWL Ontology

As concluded in Section 3.1.2, we require an OWL ontology which describes the schema of a relational database in an abstract way. This OWL representation can easily be interpreted by any remote database or application, which is capable to process OWL and has access to the Relational.OWL ontology. As a further step we use this schema representation itself as a novel ontology for creating a representation format, which is suitable for the corresponding data items.

To describe the schema of a relational database with the techniques provided by RDF and OWL, we have to define OWL classes centrally, to which any document describing such a database can refer. The abstract representation of classes like `Table` or `Column` becomes a central part of the knowledge representation process realized within OWL. Additionally, we have to specify possible relationships among these classes resulting in an ontology, a relational database can easily be described with. We call this central representation of abstract schema components and relationships the Relational.OWL ontology. It contains abstract definitions of relational databases $\mathfrak{D}$, tables $\mathfrak{T}$, columns $\mathfrak{C}$, primary keys $\mathfrak{P}$, foreign keys $\mathfrak{F}$, and its corresponding relationships.

For each relational database $\mathcal{RDB}_i$, a Semantic Web correspondent $\mathcal{ROWL}_i(\mathcal{S}_i, \mathcal{I}_i)$ is created, where $\mathcal{S}_i$ is the schema ontology and $\mathcal{I}_i$ the data instance representation. $\mathcal{S}_i$ will usually contain one subclass $\mathcal{D}_i$ of $\mathfrak{D}$. Analogously, for each relation $\mathcal{R}_1, ..., \mathcal{R}_m \in \mathcal{RDB}_i$, a subclass $\mathcal{T}_1, ..., \mathcal{T}_m$ of $\mathfrak{T}$ is created and included into $\mathcal{S}_i$. The $\in$ relationship between $\mathcal{RDB}_i$ and $\mathcal{R}_j$ is then added using a corresponding *hasTable* property within the $\mathcal{D}_i$ class. The remaining components and their relationships are transformed correspondingly. The main concepts of the Relational.OWL ontology and their corresponding relationships are represented graphically in Figure 3.2.

Similar representations based on RDF or OWL, which may evolve elsewhere, may easily be linked to the Relational.OWL ontology with corresponding `owl:equivalentClass` or `owl:equivalentProperty` relationships. As a result, database representations using one of the ontologies mapped, can be understood by any application using one of these ontologies.

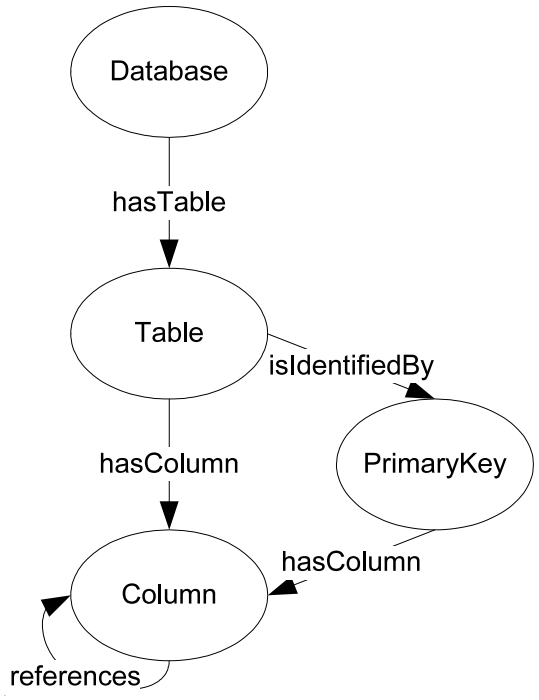

Figure 3.2: Main concepts of the Relational.OWL ontology

In other words, each component database or Semantic Web application involved in a representation based on one of these ontologies is able to process documents based on any of the interconnected representation formats. We do not even have to adapt the reasoning processes, since it is enough to create a semantic mapping between two or more ontologies to make them exchangeable, as long as they correlate semantically.

In the following we describe the components of the Relational.OWL ontology, our proposal for transforming relational data into a semantically rich representation. We can create system specific schema representations based on this ontology, which themselves can be used as ontologies for the representation of data items. In the remainder of this section we abbreviate our main namespace `http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#` with the prefix `dbs`. `Rdf`, `rdfs`, and `owl` correspond to the commonly used prefixes for RDF, RDF Schema, and OWL.

A listing of all the classes represented in the Relational.OWL ontology is provided in Table 3.1. Table 3.2 contains a list of the relationships, which interconnect these classes. The exact class and property definitions together with a small description can be accessed online at the URI specified above.

As mentioned above, we did not include a representation of all possible meta information into our ontology. Hence, items like indexes, triggers, or tablespaces

rdf:ID	rdfs:subClassOf
dbs:Database	rdf:Bag
dbs:Table	rdf:Seq
dbs:Column	rdfs:Resource
dbs:PrimaryKey	rdf:Bag

Table 3.1: Classes defined in the Relational.OWL ontology

rdf:ID	rdfs:domain	rdfs:range
dbs:has	owl:Thing	owl:Thing
dbs:hasTable	dbs:Database	dbs:Table
dbs:hasColumn	dbs:Table	dbs:Column
dbs:isIdentifiedBy	dbs:Table	dbs:PrimaryKey
dbs:references	dbs:Column	dbs:Column
dbs:length	dbs:Column	xsd:nonNegativeInteger
dbs:scale	dbs:Column	xsd:nonNegativeInteger

Table 3.2: Properties defined in the Relational.OWL ontology

are not considered, but may easily be included in a future version of Relational.OWL. In fact, the part of the relational schema we have chosen to describe is sufficient to represent the complete data stored in that database. Additional extensions in the Relational.OWL ontology would only increase the schema data overhead.

3.1.4 Schema Representation

This section provides an example on how to represent the schema of existing databases using Relational.OWL as their original ontology. The snippet in Figure 3.3 is derived from the schema representation of a corporate database containing personal information of business contacts.

The first element corresponds to the table containing the residence information of the contacts. In this case, the `rdf:ID ADDRESS` is equivalent to the table name in the original database. Instead of exclusively using the table name as an identifier, a complete URI pointing at this specific table can be specified using an identifier (cf. Section 4). Each of the five columns is defined using an `owl:DatatypeProperty` class, where all the properties required are specified.

```
<...>
 <owl:Class rdf:ID="ADDRESS">
  <rdf:type rdf:resource="&dbs;Table"/>
  <dbs:hasColumn rdf:resource="#ADDRESS.ADDRESSID"/>
  <dbs:hasColumn rdf:resource="#ADDRESS.STREET"/>
  <dbs:hasColumn rdf:resource="#ADDRESS.ZIP"/>
  <dbs:hasColumn rdf:resource="#ADDRESS.CITY"/>
  <dbs:hasColumn rdf:resource="#ADDRESS.COUNTRYID"/>
  <dbs:isIdentifiedBy>
   <dbs:PrimaryKey>
    <dbs:hasColumn rdf:resource="#ADDRESS.ADDRESSID"/>
   </dbs:PrimaryKey>
  </dbs:isIdentifiedBy>
 </owl:Class>

 <owl:DatatypeProperty rdf:ID="ADDRESS.ZIP">
  <rdf:type rdf:resource="&dbs;Column"/>
  <rdfs:domain rdf:resource="#ADDRESS"/>
  <rdfs:range rdf:resource="&xsd;string"/>
  <dbs:length>8</dbs:length>
 </owl:DatatypeProperty>

 <owl:DatatypeProperty rdf:ID="ADDRESS.COUNTRYID">
  <rdf:type rdf:resource="&dbs;Column"/>
  <rdfs:domain rdf:resource="#ADDRESS"/>
  <dbs:references rdf:resource="#COUNTRY.COUNTRYID"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
 </owl:DatatypeProperty>
</ ...>
```

Figure 3.3: Schema representation using the Relational.OWL ontology

The corresponding `&dbs;Table` and `&dbs;Column` objects are then linked using a `dbs:hasColumn` property.

The primary key property of the table is represented using a `dbs:isIdentifiedBy` property, whereas the `dbs:PrimaryKey` object corresponds to the actual primary key. Since the primary key itself may consist of more than one column, they are specified with `dbs:hasColumn` entries. The second element in Figure 3.3 describes the ZIP column of the address table, e.g. a column may contain string values with a maximum length of eight characters. The foreign key (`dbs:references`) from the `ADDRESS.COUNTRYID` column to the ID in the country table can be found in the second `owl:DatatypeProperty` element.

3.1.5 Data Representation

After having created a schema representation of a database $\mathcal{RDB}_i$ using OWL and our Relational.OWL ontology, we can regard this representation itself as a

novel ontology. With this tailored ontology-based representation of the database schema, we are able to represent the data stored in that specific database, i.e. the data can be represented as instances of its own schema ontology.

According to the possibilities given by OWL and due to the schema representation presented above, we are able to use individuals (i.e. instances) of our Relational.OWL ontology as classes. Thus the schema representation just created belongs to OWL Full, and not to OWL Lite, nor to OWL DL [DS04]. Of course, the fact that we can not restrict the complexity to one of the subclasses does not automatically result in a complex representation of data and schema items. First implementations make us confident of most RDF and OWL reasoning tools being able to handle data and schema representations created using Relational.OWL.

In order to realize this kind of data representation process, we have to ensure that all components involved (e.g. exchange partners)

- are able to process and understand RDF and OWL,
- have access to Relational.OWL or a semantically equivalent ontology, and
- have access to the schema ontology $\mathcal{S}_i$ of the corresponding database $\mathcal{RDB}_i$.

Using the schema $\mathcal{S}_i$ as a novel ontology means to represent the data instances $\mathcal{I}_i$ stored in the database $\mathcal{RDB}_i$ using a tailored data representation technique. As a result, the data can be handled using common RDF/OWL techniques for data backups, data exchanges, or any kind of data processing and reasoning tasks within the Semantic Web.

In fact, this interdigitation of schema and data corresponds exactly to the data management in relational database systems, where data items are stored as instances of their own schema. As a result, the ontology, the data is described with, changes as soon as the schema of the originating database $\mathcal{RDB}_i$ is altered. Using conventional techniques in data exchange processes would mean to manually adjust the corresponding exchange format. Using knowledge representation techniques, this is done automatically.

Besides the example presented in Section 3.1.4, where we explained how to represent the schema of a database containing information concerning business contacts, we are now able to describe the actual data stored in that specific database. In Figure 3.4 sample data items based on the schema ontology introduced in Section 3.1.4 are provided. The namespace `dbinst` points to the location where the Relational.OWL representation of the schema is stored. Hence, it is required either to hold a copy of the relevant schema file or to have access to such a representation (local copy vs. local accessible copy).

The example contains four elements, where the first two represent entries in the address table and the latter two correspond to an entry in the country table respectively. The address dataset contains all the information described in the previous example, i.e. an ID, a street, a ZIP code, a city, and a country ID. Since we are using OWL in its XML representation, we benefit from its sophisticated

```
<... />
<dbinst:ADDRESS>
 <dbinst:ADDRESS.ADDRESSID>3248</dbinst:ADDRESS.ADDRESSID>
 <dbinst:ADDRESS.STREET>Königsallee 21</dbinst:ADDRESS.STREET>
 <dbinst:ADDRESS.ZIP>40212</dbinst:ADDRESS.ZIP>
 <dbinst:ADDRESS.CITY>Düsseldorf</dbinst:ADDRESS.CITY>
 <dbinst:ADDRESS.COUNTRYID>32</dbinst:ADDRESS.COUNTRYID>
</dbinst:ADDRESS>
<... />
<dbinst:ADDRESS>
 <dbinst:ADDRESS.ADDRESSID>6824</dbinst:ADDRESS.ADDRESSID>
 <dbinst:ADDRESS.STREET>Iñigo Arista 1</dbinst:ADDRESS.STREET>
 <dbinst:ADDRESS.ZIP>31007</dbinst:ADDRESS.ZIP>
 <dbinst:ADDRESS.CITY>Pamplona</dbinst:ADDRESS.CITY>
 <dbinst:ADDRESS.COUNTRYID>152</dbinst:ADDRESS.COUNTRYID>
</dbinst:ADDRESS>
<... />
<dbinst:COUNTRY>
 <dbinst:COUNTRY.COUNTRYID>32</dbinst:COUNTRY.COUNTRYID>
 <dbinst:COUNTRY.NAME>Deutschland</dbinst:COUNTRY.NAME>
</dbinst:COUNTRY>
<... />
```

Figure 3.4: Data representation using a tailored schema ontology

features concerning internationalization: Declaring the proper encoding ensures special characters (e.g. ä, ø, or ñ) to be interpreted correctly.

Having stored all required information concerning the structure of the database in the schema file shown in Section 3.1.4, we do not need to indicate that the `ADDRESSID` corresponds to the primary key of the `dbinst:ADDRESS` table any more. The same occurs with the primary key of the `COUNTRY` table. This information is available in the accessible schema representation. It would be redundant to include it again into the data representation.

More detailed sample databases showing all features of the knowledge representation process using the Relational.OWL ontology and a larger amount of data instances can be found at `http://www.dbs.cs.uni-duesseldorf.de/RDF/`.

3.1.6 Data Overhead in the Data Exchange Process

Due to the characteristics mentioned above, especially concerning the powerful ontological meaning and reasoning implemented in OWL, we have achieved a sophisticated technique to transform relational data and schemata into a semantically rich representation. Additionally, RDF and OWL enable us to create a data and schema exchange format, which is by far more powerful than established exchange formats like genuine XML. In order to achieve this extended functionality we have to take an increased amount of data into account. However, this

increased amount of data may be a significant reason to argue against a semantic rich exchange format in data exchange processes.

```
<...>
<country>
  <column name="COUNTRYID">32</column>
  <column name="NAME">Deutschland</column>
</country>
<country>
  <column name="COUNTRYID">152</column>
  <column name="NAME">España</column>
</country>
</ ...>
```

Figure 3.5: XML data generated with `Rec2XML`

For testing how much data overhead is produced with our technique, we created a cutout of a Data Warehouse with synthetic data containing one table (the fact table) with six columns. We first extracted the data using the Relational.OWL application (cf. Section 5.1.1) and compared it with the results of the `Rec2XML` scalar function in IBM's UDB which generates XML data from a relational query without any semantic representation (cf. Figure 3.5). The result revealed a certain data overhead of Relational.OWL (Figure 3.6) compared to `Rec2XML`. However, this linear increase of data is plausible since our approach includes additional knowledge representation in each data item, lacking in the plain XML format.

Although the resulting overhead seems to be a major drawback it is not that substantial. Further evaluations have revealed that data represented with Relational.OWL can be compressed with a higher ratio than `Rec2XML` (cf. Figure 3.6). After a compression, both representations differ very little in their corresponding file sizes. This data overhead is thus only a minor drawback taking into account the remarkable benefit achieved through the semantic representation.

3.2 Relational.OWL and RDF-QL

In Section 3.1 we presented Relational.OWL, a technique to extract the semantics of a relational database and transform it into RDF/OWL, a machine readable and understandable format. For this purpose, the schema information of the data source is extracted and converted into an ontology. Afterwards, the data items are represented as instances of this data source specific ontology.

In this section we analyze, whether the combination of Relational.OWL as a Semantic Web representation of virtually any relational database and RDF query language (RDF-QL) like RDQL [Sea04] or its successor SPARQL [PS05] could replace existing, usually isolated, relational to semantic mappings. Indeed, comparing the expressiveness of a premature query languages like RDQL or SPARQL

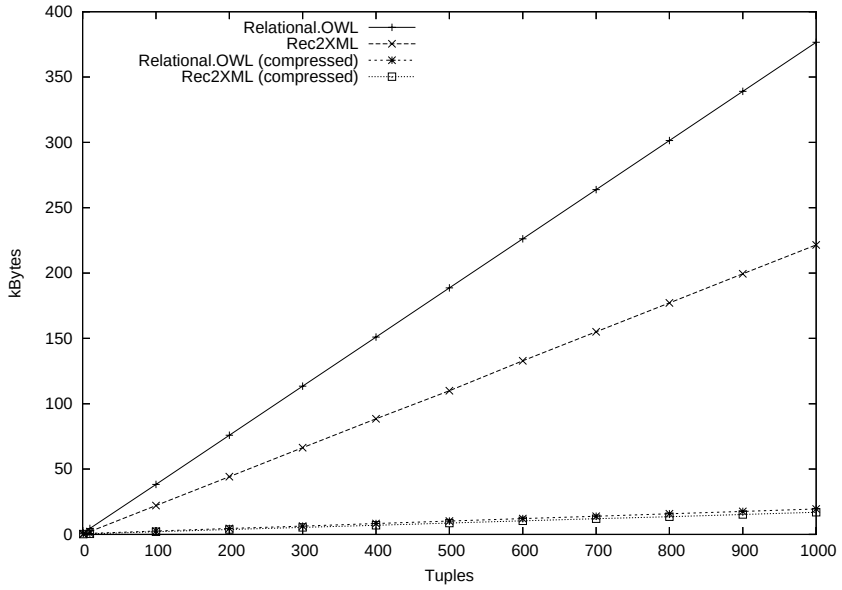

Figure 3.6: Amount of data after a data export

with a sophisticated language like SQL, which has developed throughout the decades, would rather be unfair. We thus have decided to limit our evaluation to the basic operations of the relational algebra. In other words, we want to analyze in this section, whether RDQL or SPARQL are relational complete or not. Since there is no formal foundation for these query languages, and a supplementary formalization leads to several mismatches (cf. [Cyg05]), both evaluations have to be performed on a more pragmatic level.

3.2.1 Relational.OWL and RDQL

Having created an automatic transformation mechanism from data stored in relational databases into a representation, which is processable by basically any Semantic Web application, all kinds of legacy data stored in relational databases become an integral part of the Semantic Web.

As a result, Semantic Web applications needing access to data stored in relational databases do not have to query these databases using relational query languages any more. They may equally use their preferred query language like RQL [KCPA01], RDQL [Sea04], or Xcerpt [BS02], as long as this query language provides the required expressiveness. In this section we analyze RDQL as a representative for all RDF query languages, since it is supported by the Jena

Framework [Jen06] and submitted to the W3C [Sea04]. All queries presented in this section have been verified using the Jena implementation of RDQL. Nevertheless, any RDF query language could be evaluated accordingly (cf. Section 3.2.2).

We analyze whether all the possible queries on the original relational database can be expressed using RDQL on the Relational.OWL representation of that specific database. In fact SQL has developed throughout the years from a simple query language based on the relational calculus to a powerful language for integrating data from across multiple data sources (cf. [MMJ+01b]). Hence, we have decided to compare the expressiveness of RDQL only with the relational algebra [Cod72] and not with SQL, i.e. to check if RDQL is relational complete or not.

The comparison is based on the sample database containing personal and contact information of business partners already introduced in Section 3.1. It contains the following two relations:

`Address(AddressID, Street, ZIP, City, CountryID)` and
`Country(CountryID, Name)`.

Since there are various positions on how to verify the relational completeness of a query language (cf. [CH79]), we have decided to follow the perception of Elmasri and Navathe [NE01] regarding the set $\{\sigma, \pi, \cup, -, \times\}$ of relational operations as complete. Additionally we show how to realize the join operation with RDQL, since it is one of the most important operations of relational queries. Due to the fact that RDQL is not closed, i.e. the result of an RDQL query is not an RDF triple but a list of possible variable bindings, a direct comparison to the relational algebra, which itself is closed, may in some cases be slightly imprecise.

3.2.1.1 Selection

One of the basic operations of the relational algebra is the selection σ. The expression

$$\sigma_{Name="Australia"}(r(Country)) \tag{3.1}$$

thus selects all tuples of the `Country` relation where the attribute `Name` equals `Australia`. Since we have created an own `OWL:Class` for each relation in our database, we have to apply a similar constraint for the objects of this class to obtain the corresponding result with the Semantic Web version of our database. A possible RDQL query hence is

```
SELECT  ?x, ?y, ?z
WHERE   (?x, rdf:type, dbinst:COUNTRY)
        (?x, dbinst:COUNTRY.NAME, "Australia")
        (?x,?y,?z)
USING   dbinst for [...]
        rdf for [...]
```

The RDQL query representing the selection contains three main clauses. Since RDQL is not closed, we have decided to include the three variables into the `SELECT` clause, from which a valid RDF triple could be generated. In the first line of the `WHERE` clause we restrict the result set to contain only objects of the type `dbinst:COUNTRY` having their origin in the `Country` relation of our database. The actual selection σ is performed in the next line, where we enforce the value of the property `dbinst:COUNTRY.NAME` of all the objects represented by the `?x` variable to be `Australia`. The last line of the `WHERE` clause is required to select the entire set of triples describing the classes which fulfil the conditions described above. Both, the `rdf` and the `dbinst` prefixes are defined in the `USING` clause, representing the commonly used prefix for RDF and the URI for the schema of the database respectively. Since we need the same prefix definitions in all remaining RDQL queries, we do not describe them in the following sections once again. Please note, that besides the selection presented in this section, we are also able to represent selections containing inequality operations like $<$ or $>$ using an additional variable and constraining its values in the `AND` clause (e.g. `AND ?a > 25`).

3.2.1.2 Projection

We are able to select relevant attributes of a relation with the projection operation π. Hence, the following expression means that the `Street` and `City` attributes are picked out of the `Address` relation:

$$\pi_{Street,City}(r(Address)). \tag{3.2}$$

Unlike SQL, we cannot use the `SELECT` clause of RDQL for the projection. It has to be done in the `AND` clause where we can specify more complex constraints.

```
SELECT  ?x, ?y, ?z
WHERE   (?x, rdf:type, dbinst:ADDRESS)
        (?x,?y,?z)
AND     ((?y EQ dbinst:ADDRESS.STREET)||
        (?y EQ dbinst:ADDRESS.CITY))
USING   dbinst for [...]
        rdf for [...]
```

Analogous to the query described above, the result set is restricted to objects of the `dbinst:ADDRESS` type in the `WHERE` clause. The actual projection is done in the `AND` part of the query, where we require the properties of the result triples (i.e. `?y`) to be either `dbinst:ADDRESS.STREET` or `dbinst:ADDRESS.CITY`. The result is a list of all the triples containing city or street information within an address object.

3.2.1.3 Set Union

The union $\cup$ operation unifies two union-compatible relations (cf. [Dat82]). The expression

$$\pi_{CountryID}(r(Address)) \cup \pi_{CountryID}(r(Country)) \tag{3.3}$$

thus unifies all tuples from the `CountryID` attribute in the `Address` relation with those of the `Country` relation. If we want to query the Semantic Web representation of the database using RDQL, we first have to perform the projection within the `AND` clause restricting the `?y` variable to both `COUNTRYID` attributes. The restriction to both classes is done in the remaining two lines of the `AND` clause.

```
SELECT  ?x, ?y, ?z
WHERE   (?x, ?y,?z)
        (?x,rdf:type,?a)
AND     ((?y EQ dbinst:COUNTRY.COUNTRYID)||
         (?y EQ dbinst:ADDRESS.COUNTRYID))&&
        ((?a EQ dbinst:COUNTRY)||
         (?a EQ dbinst:ADDRESS))
USING   dbinst for [...]
        rdf for [...]
```

This RDQL query thus returns all `COUNTRYID`s originated in both, the `dbinst:COUNTRY` and `dbinst:ADDRESS` objects, i.e. the same as our expression of the relational algebra.

3.2.1.4 Set Difference

If it is required to obtain all the tuples contained in one relation and not in a second one, we use the set difference $-$. Thus, the expression

$$\pi_{CountryID}(r(Country)) - \pi_{CountryID}(r(Address)) \tag{3.4}$$

returns all existing `CountryID`s not used in the `Address` relation. The projection has been introduced only to obtain union-compatibility (cf. [Dat82]).

Within the corresponding RDQL query, objects of the type `dbinst:COUNTRY` are represented by the variable `?a` and the `dbinst:ADDRESS` objects by `?x`. The set difference constraint is specified in the `AND` clause where we refer to the values of both `COUNTRYID` properties assigned to the variables in the `WHERE` clause.

```
SELECT  ?b
WHERE   (?a, dbinst:COUNTRY.COUNTRYID, ?b)
        (?a, rdf:type, dbinst:COUNTRY)
        (?x, dbinst:ADDRESS.COUNTRYID, ?y)
        (?x, rdf:type, dbinst:ADDRESS)
```

```
AND     !(?b EQ ?y)
USING   dbinst for [...]
        rdf for [...]
```

Similar to the queries presented above, this RDQL query returns exactly the same information as its corresponding relational algebra expression.

3.2.1.5 Cartesian Product

The Cartesian product $\times$ unifies two relations into a new relation containing the complete set of attributes from the two original relations. The values of this relation are a combination of all tuples of the first relation with all tuples of the second relation. The expression

$$r(Country) \times r(Address) \tag{3.5}$$

thus corresponds to a relation containing all attributes from the `Country` relation and all those from the `Address` relation. The original attributes are renamed to guarantee their uniqueness (cf. [Dat82]). The amount of values corresponds to $(m * n)$, whereas m is the number of values in the first table and n in the second table respectively.

The definition of a Cartesian product within the Semantic Web is more complex than it seems at a first glance. Melnik, for example, does not mention a Cartesian product of RDF triples or Semantic Web objects within his RDF algebra [Mel99]. Intuitively, the Cartesian product of two sets with m and n objects would be to create $(m * n)$ new objects, containing the properties of two objects, one of each set respectively (cf. [FHVB04]).

Since RDQL is not closed and we cannot receive objects as a result form an RDQL query, we have to express the Cartesian product differently. There are two main options on how to express the Cartesian product. Both are as close as possible to the Cartesian product of the relational model.

The first option returns all possible combinations of two properties, each from a different set of objects, i.e. one property from the `dbinst:COUNTRY` and one from the `dbinst:ADDRESS` objects at a time:

```
SELECT  ?a, ?b, ?c, ?x, ?y, ?z
WHERE   (?a, ?b, ?c)
        (?a, rdf:type, dbinst:COUNTRY)
        (?x, ?y, ?z)
        (?x, rdf:type, dbinst:ADDRESS)
USING   dbinst for [...]
        rdf for [...]
```

The second option returns a list of all properties contained in any object of the `dbinst:COUNTRY` and `dbinst:ADDRESS` classes.

```
SELECT  ?x, ?y, ?z
WHERE   (?x, ?y,?z)
        (?x,rdf:type,?a)
AND     ((?a EQ dbinst:COUNTRY)||
         (?a EQ dbinst:ADDRESS))
USING   dbinst for [...]
        rdf for [...]
```

Intuitively, this query seems to be more adequate than the one mentioned above. However, it is very similar to the RDQL query in Section 3.2.1.3 where we expressed the set union. The main difference between both queries is only the restriction in the union query.

3.2.1.6 (Equi-)Join

The most important relational operation is indeed the join operation $\bowtie$ introduced in [Cod79]. The θ join of two relations R_1 and R_2 relating to their attributes B_1 and B_2 is the concatenation of the attributes of R_1 and R_2, including their corresponding values, whenever attribute B_1 and B_2 correlate with the θ condition. If θ is $=$, the join operation is called *equi-join*. Since the join operation is usually stated in terms of the Cartesian product (cf. [NE01]), the translation of the join operation to RDQL may help to decide, which of both possibilities described in Section 3.2.1.5 should be considered the Cartesian product RDQL correspondent.

The two relations `Address` and `Country` can be joined with the expression

$$r(Address) \bowtie_{CountryID=CountryID} r(Country). \tag{3.6}$$

Contrary to the natural join, the resulting relation contains all the attributes from the first and the second relation including both `CountryID` attributes.

Once again, since RDQL is not complete, we cannot find an exact equivalent query to the expression of the relational algebra just mentioned. However, we can express quite similar constraints and put both `dbinst:COUNTRY` and `dbinst:ADDRESS` objects into a corresponding relation.

```
SELECT  ?a, ?d, ?e
WHERE   (?a, ?d, ?e)
        (?a, rdf:type, ?c)
        (?x, rdf:type, dbinst:COUNTRY)
        (?x, dbinst:COUNTRY.COUNTRYID, ?y)
        (?r, rdf:type, dbinst:ADDRESS)
```

```
        (?r, dbinst:ADDRESS.COUNTRYID, ?s)
AND     (((?c EQ dbinst:COUNTRY) || (?c EQ dbinst:ADDRESS)) &&
         (?y EQ ?s) &&
         ((?x EQ ?a) || (?r EQ ?a)))
USING   dbinst for [...]
        rdf for [...]
```

For expressing the required join condition between both classes, we first define a free result variable `?a`. The objects of the `dbinst:COUNTRY` class are bound to `?x` and those of the `dbinst:ADDRESS` class to `?r`. The values of the relevant `COUNTRYID` attributes are bound to `?y` and `?s` correspondingly. The remaining relation between these bound and unbound variables is specified in the `AND` clause, where we restrict the result set to either be a `dbinst:COUNTRY` or a `dbinst:ADDRESS` object. The actual equality condition for the values in `?y` and in `?s` from the join condition is given in the next line. The free variable `?a` is finally bound to the result set in the last line of the `AND` clause.

The decision, which of the queries described in Section 3.2.1.5 should be considered as the RDQL correspondent of the Cartesian product remains unanswered. Even though the query defined in this section certainly signs to the second alternative, where we also had a free variable, this question may probably not be answered precisely until the queries of RDQL can be referred to as closed.

3.2.1.7 Discussion

In this section we analyzed whether a Semantic Web representation of a relational database could potentially replace existing interfaces for the access of relational data out of the Semantic Web. The advantage of such an approach is obvious: All Semantic Web applications could query data stored in relational databases using their own built-in query languages without having to convert this data in a manual and time-consuming process. A direct comparison of this approach with SQL, the commonly used interface to relational databases, would be unfair, since SQL has evolved during the last decades and most semantic query languages are rather rudimental. Hence, we decided to check whether the expressiveness of querying semantic data created with Relational.OWL is similar to that of the basic relational algebra. We chose RDQL as an exemplary query language since it is widely accepted and provides a stable implementation.

During our analysis we observed that the main problem of RDQL (and of most semantic query languages) is the absence of closeness, i.e. the result set of a query is not a valid RDF/OWL expression any more.

Although we were able to simulate most of the main relational algebra operations, we could not find a clear representative for the Cartesian product. As long as this fundamental drawback is not resolved, we probably will not achieve a language capable to compete with SQL.

3.2.2 Relational.OWL and SPARQL

Since we did not reach a satisfactory conclusion in Section 3.2.1 concerning the relational completeness of RDQL and Relational.OWL, we now analyze SPARQL [PS06b] as an RDF query language representative. SPARQL is the successor of RDQL, eliminating many of its drawbacks like lack of expressiveness and completeness. Currently declared as a candidate recommendation, it will probably soon be recommended as a de facto standard by the W3C (cf. [Jac05]). Despite its novelty, the Jena Framework [Jen06] already supports SPARQL using its ARQ extension. All queries presented in this section have been verified using this implementation of SPARQL.

According to the argumentation of Section 3.2.1, we have again decided to limit this comparison to the most important expressions of SQL, which can be represented by the basic operations of the relational algebra [Cod72].

The comparison is again based on the simple database containing personal and contact information of business partners with the two relations:

`Address(AddressID, Street, ZIP, City, CountryID)` and
`Country(CountryID, Name)`.

Since there are various positions on how to verify the relational completeness of a query language (cf. [CH79]), we have again decided to follow the perception of Elmasri and Navathe [NE01] regarding the set $\{\sigma, \pi, \cup, -, \times\}$ of relational operations as complete and show additionally, how to realize the (equi-)join operation with SPARQL.

As we have seen in Section 3.2.1, using a non-closed language like RDQL leads to insuperable problems regarding the simulation of more complex operations like the set union $\cup$. We thus have decided to use the `CONSTRUCT` and `DESCRIBE` and not the `SELECT` instructions of SPARQL to make sure, that all responses to our queries are valid RDF, i.e. closed. For a detailed introduction to the query language SPARQL we refer to [PS05].

3.2.2.1 Selection

One of the basic operations of the relational algebra is the selection σ (not to be mistaken with the `SELECT` clause within an SQL query). The expression

$$\sigma_{Name=\text{"}Australia\text{"}}(r(Country)) \tag{3.7}$$

thus selects all tuples of the `Country` relation where the attribute `Name` equals the term `Australia`. Since we have created an own `OWL:Class` for each relation in our database, we have to apply a similar constraint for the objects (or nodes)

of this class to obtain the corresponding result with the Semantic Web version of our database. A possible SPARQL correspondent is

```
PREFIX   rdf:[...]
PREFIX   dbinst:[...]
DESCRIBE ?a
WHERE    {{?a rdf:type dbinst:COUNTRY} .
          {?a dbinst:COUNTRY.NAME 'Australia'}}
```

The SPARQL query representing the selection contains three main clauses. The first two lines, containing a `PREFIX` clause each, define the `rdf` and `dbinst` prefixes, representing the commonly used namespace of RDF and the URI for the schema of the corresponding database. Since the definitions of these prefixes will be required throughout this document, it will not be described in the following sections once again.

With the `DESCRIBE` clause of SPARQL we are able to abbreviate those queries representing simple nodes. `DESCRIBE ?a` means, that the result set contains the objects represented by the variable `?a` including all their corresponding properties. In the first line of the `WHERE` clause we restrict the result set to contain only objects of the type `dbinst:COUNTRY`, having their origin in the `Country` relation of our database. The actual selection σ is performed in the next line, where we enforce the value of the property `dbinst:COUNTRY.NAME` of all the objects represented by the `?a` variable to be `Australia`.

The result set of this query contains all objects of type `dbinst:COUNTRY`, where the value of the `dbinst:COUNTRY.NAME` property is `Australia`. The advantage of the `DESCRIBE` clause is, that we can ensure all the properties of the matching objects to be part of the corresponding objects within our result set.

3.2.2.2 Projection

Relevant attributes within a relation can be extracted with the projection operation π. Hence, the following expression means that both, the `Street` and the `City` attributes are picked out of the `Address` relation:

$$\pi_{Street,City}(r(Address)). \tag{3.8}$$

Unlike SQL, we neither can use the `SELECT`, nor the `CONSTRUCT` or `DESCRIBE` clauses of a SPARQL query for the projection. It has to be done in a `FILTER`

construct within the WHERE clause, where more complex constraints can be specified.

```
PREFIX    rdf:[...]
PREFIX    dbinst:[...]
CONSTRUCT {?a ?b ?c}
WHERE     {{?a ?b ?c} .
           {?a rdf:type dbinst:ADDRESS} .
      FILTER (?b=dbinst:ADDRESS.STREET
              ||?b=dbinst:ADDRESS.CITY)}
```

In Section 3.2.2.1 we were able to use the DESCRIBE clause of SPARQL. In this case, it is not possible, since we do not want to have all the properties of the matching objects to be represented in the result set, but only those specified within the FILTER clause, i.e. the properties dbinst:ADDRESS.STREET and dbinst:ADDRESS.CITY of the dbinst:ADDRESS objects.

To achieve our goal of creating result objects containing only the required dbinst:ADDRESS.STREET and dbinst:ADDRESS.CITY properties, we have to provide three attributes in the CONSTRUCT clause, representing the triples from which a corresponding object can be created. These variables are bound in the first line of the WHERE clause, representing the entire set of triples, which describe the objects fulfilling the conditions specified in the WHERE clause.

In this case, the result set consists of *untyped* objects containing only the specified attributes. We have intentionally omitted the rdf:type properties to emphasize our goal of selecting only the attributes specified in the relational algebra expression. Certainly, the rdf:type properties could easily be added specifying it either in the FILTER construct or in the CONSTRUCT clause.

3.2.2.3 Set Union

The union $\cup$ operation unifies two union-compatible relations (cf. [Dat82]). The expression

$$\pi_{City}(r(Address)) \cup \pi_{Name}(r(Country)) \tag{3.9}$$

thus unifies all tuples from the City attribute in the Address with the Name tuples of the Country relation.

As an extension to RDQL, SPARQL provides the UNION construct to unify two sets of objects, which exactly corresponds to the signification in the relational algebra. Unlike the relational algebra, SPARQL does not demand both sets of objects to be union-compatible, this may either be intentionally or because of the not yet finished standardization process of the SPARQL recommendation. Nevertheless, we have added this additional constraint to our SPRAQL query, in order to match the query specified above.

After the usual PREFIX definitions, a blank node with property ?b and corresponding value ?c is created for each result. The property either comes from the dbinst:ADDRESS objects or from those of the dbinst:COUNTRY class. Therefore, the ?b and ?c variables are bound to the dbinst:ADDRESS.CITY and the dbinst:COUNTRY.NAME properties and to their corresponding values.

```
PREFIX    rdf:[...]
PREFIX    dbinst:[...]
CONSTRUCT {_:v ?b ?c}
WHERE     {{?a ?b ?c} .
          {{?a dbinst:ADDRESS.CITY ?c} .
          {?a rdf:type dbinst:ADDRESS}}
          UNION
          {{?a dbinst:COUNTRY.NAME ?c} .
          {?a rdf:type dbinst:COUNTRY}}}
```

This SPARQL query returns all COUNTRY.NAMEs and ADDRESS.CITYs within blank nodes. Thus, each object corresponds to a tuple within the original result set of the relational algebra expression.

3.2.2.4 Set Difference

If it is required to obtain all the tuples contained in one relation and not in a second one, we use the set difference $-$. Thus, the expression

$$\pi_{CountryID}(r(Country)) - \pi_{CountryID}(r(Address)) \tag{3.10}$$

returns all existing CountryIDs not used in the Address relation. The projection has been introduced to obtain union-compatibility (cf. [Dat82]).

Within the corresponding SPARQL query, objects of the type dbinst:COUNTRY are represented by the variable ?a and the dbinst:ADDRESS objects by ?b. Unfortunately, SPARQL does not provide an inverse to the UNION operation supported, hence we have to simulate it. The set difference constraint is specified in the FILTER construct within the WHERE clause, where we refer to the values of both COUNTRYID properties assigned to the ?c and ?d variables.

```
PREFIX    rdf:[...]
PREFIX    dbinst:[...]
CONSTRUCT {_:v dbinst:COUNTRY.COUNTRYID ?c}
WHERE     {{?a rdf:type dbinst:COUNTRY} .
          {?a dbinst:COUNTRY.COUNTRYID ?c} .
          {?b rdf:type dbinst:ADDRESS} .
          {?b dbinst:ADDRESS.COUNTRYID ?d }.
          FILTER (?c != ?d)}
```

As a result, all dbinst:COUNTRY.COUNTRYIDs, which do not appear within an dbinst:ADDRESS object, are part of the result set. Each COUNTRYID is represented within a blank node.

3.2.2.5 Cartesian Product

The Cartesian product $\times$ unifies two relations into a new one, containing the complete set of attributes from the two original relations. Its values are a combination of all tuples of the first relation with all tuples of the second relation. The expression

$$r(Country) \times r(Address) \tag{3.11}$$

thus corresponds to a relation containing all attributes from the Country relation and all those from the Address relation. The original attributes are renamed to guarantee their uniqueness (cf. [Dat82]). The amount of values is $(m*n)$, whereas m and n correspond to the number of values in the first and second relations respectively (cf. Section 3.2.1.5).

The main advantage of SPARQL in comparison to RDQL, the possibility to create RDF models out of a query, has its biggest impact on the simulation of this operation. Had it been impossible to create an unambiguous correspondent to the Cartesian product with RDQL, we can simulate this operation with SPARQL. Unlike in the relational algebra, the Cartesian product simulation mentioned above does not result in $(m*n)$ new objects, but in m objects, each containing one object of the dbinst:COUNTRY class and all the objects from the dbinst:ADDRESS class. This result may surprise at first glance, since we would expect $(m * n)$ objects to be created. If we regard the Cartesian product as a nested loop, where each dbinst:COUNTRY object, the attributes of each dbinst:ADDRESS is added, the result makes sense, since the attributes added to the dbinst:COUNTRY objects are not added to a copy of that object, but to that object itself.

As a result, the Cartesian product simulated with Relational.OWL and SPARQL is not commutative, i.e.

$$r(Country) \times r(Address) \neq r(Address) \times r(Country). \tag{3.12}$$

Although this means to impose an important restriction to the Cartesian product, we are nevertheless able to simulate this operation with:

```
PREFIX    rdf:[...]
PREFIX    dbinst:[...]
CONSTRUCT {?a ?b ?c;
               ?e ?f}
WHERE     {{?a ?b ?c;
               rdf:type dbinst:COUNTRY} .
           {?d ?e ?f;
               rdf:type dbinst:ADDRESS}}
```

Since each node of this result set contains properties from objects of both classes, it also contains two `rdf:type` properties: one with `dbinst:COUNTRY` and the second with `dbinst:ADDRESS` as its value. The resulting nodes are thus objects of both types. This characteristic of the newly created nodes is intended, since the Cartesian product of two objects intuitively corresponds to the multiple inheritance or composition operations known from the object oriented software design.

3.2.2.6 (Equi-)Join

Even though all remaining operations of the relational algebra can be composed with the basic operations of the relational algebra mentioned above, we additionally want to translate the most important relational operation join ($\bowtie$) (cf. [Cod79]) into its SPARQL correspondent.

The θ join of two relations R_1 and R_2 relating to their attributes B_1 and B_2 is the concatenation of the attributes of R_1 and R_2, including their corresponding values, whenever attribute B_1 and B_2 correlate with the θ condition. If θ is $=$, the join operation is called *equi-join*.

The two relations `Address` and `Country` can be joined with the expression

$$r(Address) \bowtie_{CountryID=CountryID} r(Country). \tag{3.13}$$

Contrary to the natural join, the resulting relation contains all the attributes from the first and the second relation, including both `CountryID` properties. The table names for both `CountryID`s in equations 3.13 and 3.14 have been omitted for a better readability. The corresponding attributes should be stated as `Address.CountryID` and `Country.CountryID` respectively.

Since the join operation itself is not part of the basic operations of the relational algebra, it can be stated in its terms, e.g. using the Cartesian product and the selection (cf. [NE01]). Thus, we may express the equi-join also with the following expression:

$$\sigma_{CountryID=CountryID}(r(Address) \times r(Country)). \tag{3.14}$$

The corresponding SPARQL query should then either be described using nested SPARQL queries or by a combination of the selection and Cartesian product operations. Since the SPARQL candidate recommendation [PS06b] does not provide the possibility to express nested queries, we have to query the original RDF/OWL model at first with a Cartesian product. The resulting RDF graph is then queried with the additional selections required. We have decided to base our (equi-)join representation on a combination of the selection and Cartesian product operations, since it consists of a single SPARQL query. Nevertheless, the Cartesian product and selection queries required for a corresponding nested query simulation may easily be given.

Analogous to the relevant queries stated in the previous sections, the prefixes `rdf` and `db` are defined in the first lines of the SPARQL query given below. Both variables `?a` and `?d` are again bound to the `dbinst:COUNTRY` and `dbinst:ADDRESS` objects respectively, resulting in a Cartesian product (cf. Section 3.2.2.5). The additional constraint imposed by the equi-join is achieved in the last two lines, where the values of both `COUNTRYID` properties are specified to be equal, corresponding to the selection operation already described in Section 3.2.2.1.

```
PREFIX    rdf:[...]
PREFIX    dbinst:[...]
CONSTRUCT {?d ?b ?c;
              ?e ?f}
WHERE     {{?a ?b ?c;
               rdf:type dbinst:COUNTRY} .
           {?d ?e ?f;
               rdf:type dbinst:ADDRESS} .
           {?a dbinst:COUNTRY.COUNTRYID ?x} .
           {?d dbinst:ADDRESS.COUNTRYID ?x}}
```

Similar to the execution of the $\times$ operation in Section 3.2.2.5, the result set of this SPARQL query is not commutative and contains objects, which are simultaneously of the `dbinst:ADDRESS` and the `dbinst:COUNTRY` type. Anyhow, since the resulting objects contain all the properties of both originating objects, the query fulfills the equi-join constraints of the relational algebra.

3.2.2.7 Discussion

Following the results of Section 3.2.1, where we discovered some fundamental drawbacks in the query language RDQL, especially concerning its closeness, we have decided to perform the same analysis choosing SPARQL as an RDF query language representative, since it is expected to be recommended as a de facto standard by the W3C soon.

During our analysis, we observed the lack of some basic operations like `MINUS` in the current version of the SPARQL recommendation. Nevertheless and despite the vague introduction of the `UNION` operation and the surprising results with the Cartesian product, we actually managed to simulate the basic operations $\{\sigma, \pi, \cup, -, \times\}$ of the relational algebra. Additionally we showed, that join operations, which will certainly be part of most mapping operations between the relational and the semantic worlds, can easily be deduced from our Cartesian product simulation — just as it is done within the relational algebra. Since we were successfully able to simulate more than the basic relational operations, we

have shown, that the combination of Relational.OWL and SPARQL are relational complete.

As we have seen, we are in a position, where we can replace existing self-made relational database to Semantic Web bridges with a Relational.OWL representation of the database together with a query language like SPARQL. In fact, the expressiveness of SPARQL goes beyond those operations analyzed in this work. Although SPARQL still lacks grouping and update functionalities and does not support nested queries, its expressiveness approaches, with operations like `DISTINCT`, `ORDER BY`, or `LIMIT`, that of basic SQL. Although we are aware, that our approach will unlikely replace all existing SQL gateways from relational databases to the Semantic Web, we have shown, that it certainly could.

3.3 Relational to Semantic Mapping

Despite being processable by any application understanding RDF and OWL, the data extracted using Relational.OWL still lacks *real* semantic meaning. Indeed, the information originally stored in relational tables is represented within a table object and not within an appropriate Semantic Web object, e.g. an `http://www.w3.org/2000/10/swap/pim/contact#Person` object. This drawback has to be accepted in order to achieve an automatic transformation from relational databases to the Semantic Web.

Nevertheless, many applications still require the data to be represented as *real* semantic objects for being able to perform reasoning or further data processing tasks. To meet the demands of such applications, a data mapping from the relational to the required data representation is needed.

3.3.1 Requirements

Existing approaches like [BS04] introduce a special mapping language, which although it is often based on RDF, still has to be understood and adopted by all administrators needing to perform a single mapping from a relational database to the Semantic Web. The technique presented in this section goes one step further and uses common RDF query languages for the mapping task. The following requirements have to be kept to use such query languages as a mapping language.

Relational.OWL: Contrary to a common mapping, where the relational data is directly translated into the Semantic Web, our approach passes one additional step. First, we represent the data stored in the original relational database in a semantic-rich format, i.e. in RDF/OWL. This step is either done exporting the complete data and schema sets into RDF using the Relational.OWL application (cf. Section 5.1) or using the virtual database representation provided by RDQuery (cf. Section 5.2). Please note, that both

data transformation methods are processed automatically without any human intervention. Both techniques result in a Semantic Web representation of the data and schema components of the original relational database.

Closed Query Language: We are potentially able to perform a mapping using any of the upcoming query languages, as long as it is closed within RDF and contains a construct similar to the `CONSTRUCT` clause in SPARQL (cf. [PS06b]). Otherwise the resulting variable bindings would have to be translated again into RDF. We have chosen SPARQL as a representative query language, since it is easy to understand, its syntax is based on SQL, it is as powerful as the relational algebra in its expressiveness (cf. Section 3.2.2).

Target Ontology: Although the Relational.OWL representation (cf. Section 3.3.2) of the database is processable by virtually any Semantic Web application, it still lacks *real* semantics, since the data is represented as it was stored in the relational database, i.e. stored in tables and columns. Since we want to assign this data a real meaning, we require a target ontology, it can be mapped to.

3.3.2 Definitions

In this section we define the basic terms used for our relational database to Semantic Web mapping approach. First, we introduce the *semantic translation* of relational databases into the Semantic Web:

Definition 1 (Semantic Translation)
The semantic translation $\mathcal{ST}(\mathcal{RDB}, \mathcal{ROWL})$ of a relational database $\mathcal{RDB}$ into its Relational.OWL representation $\mathcal{ROWL}$ (see Definition 2 below) is an automatic translation process, where for each $\mathcal{RDB}$ and its schema components, a Semantic Web correspondent is created (cf. Section 3.1).

In this context, the *Relational.OWL representation* of a database is:

Definition 2 (Relational.OWL Representation)
The Relational.OWL representation of a relational database is described by $\mathcal{ROWL}(\mathcal{S}, \mathcal{I})$, where $\mathcal{S}$ is the schema representation of the database as seen in Section 3.1 and $\mathcal{I}$ contains the corresponding data instances of the schema components described with $\mathcal{S}$.

Having created a Relational.OWL representation of the corresponding database, we are now able to perform a *mapping*:

Definition 3 (Mapping)
A mapping $\mathcal{M}$ from a relational database to the Semantic Web is a four-tuple $\mathcal{M}(\mathcal{RDB}, \mathcal{ROWL}, \mathcal{TO}, \mathcal{Q})$, with $\mathcal{RDB}$ being the source database, $\mathcal{ROWL}$ the

Relational.OWL representation of $\mathcal{RDB}$, $\mathcal{TO}$ the target ontology, and $\mathcal{Q}$ the mapping query, expressed in a (closed) query language $\mathcal{QL}$.

Contrary to the *Relational.OWL representation* created with an automatic semantic translation, a *mapping* has to be stated manually using a query $\mathcal{Q}$. The mapping is correct, iff querying $\mathcal{ROWL}$ with $\mathcal{Q}$ results in one ore multiple instances of $\mathcal{TO}$. Hence $\mathcal{Q}$ has to fulfill two main properties. First, it has to be adequate in regard to $\mathcal{ROWL}$, i.e. return the desired result and secondly, the resulting data items have to be formatted as instances of $\mathcal{TO}$.

3.3.3 Mapping Process

The complete relational data to RDF mapping process is illustrated in Figure 3.7. It consists of two main steps, which were already introduced in the previous sections.

First, the Relational.OWL representation of the schema and the data components of the original data source are generated. The schema representation becomes thereby an instance of the Relational.OWL ontology. In turn, the data items converted become instances of the schema ontology just created. This step could either be performed using the Relational.OWL application (cf. Section 5.1), i.e. the schema and data components are translated statically in a one-time process, or using a virtual representation of that RDF model, e.g. with RDQuery (cf. Section 5.2). The advantage of the latter is obvious, since the data stock, on which the queries are performed, is always up-to-date. This cannot be guaranteed using the Relational.OWL application. Nevertheless, if the source database does not change frequently, a static translation into the Relational.OWL representation could be enough.

Having created the Relational.OWL representation of the relational database, the second step including the actual mapping can be performed.

The RDF model just created may now be queried with an arbitrary RDF query language. As long as the query language is closed, the resulting query response is again within the Semantic Web, i.e. it is a valid RDF model or graph and may then be processed by other Semantic Web applications using their own built-in functionality.

Using the `CONSTRUCT` clause of a query language like SPARQL (cf. [PS06b]), the resulting data items can be inserted into an arbitrary RDF skeleton. This property of the query language is vaguely comparable to an XSLT-Stylesheet [XSL99]. If we specify an adequate RDF skeleton, we can achieve the resulting RDF model to correspond to an instance of the intended target ontology. The RDF skeleton in the `CONSTRUCT` clause of the SPARQL query becomes hereby the pivotal part of the actual mapping process. A sample mapping query is provided in Section 3.3.6.

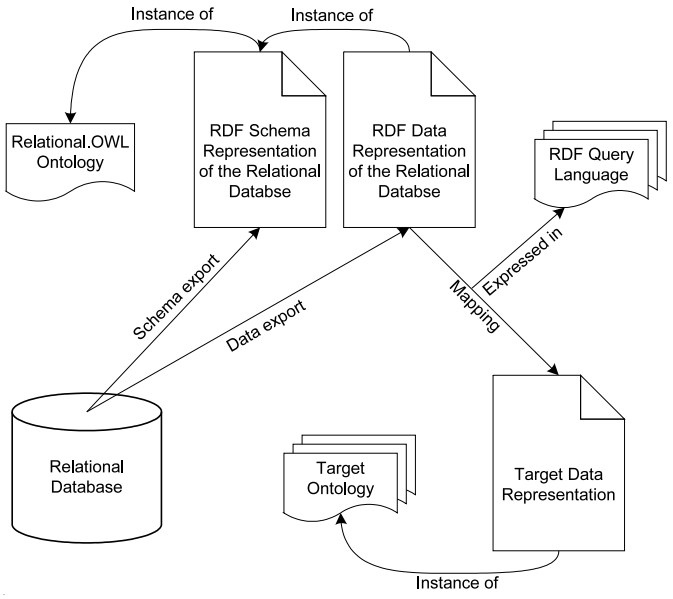

Figure 3.7: Mapping process

3.3.4 Characteristics

In this section we discuss the major characteristics of our relational database to RDF/OWL mapping approach.

Combination of Automatic and Manual Mappings: The mapping approach presented in this section is suitable for most relational database scenarios. If we have to handle with constantly changing database schemas, an automatic mapping with Relational.OWL into the Semantic Web is the best choice. Indeed, an automatic mapping with Relational.OWL does not add real semantics to the RDF objects, but at least, the data is processable by most Semantic Web application without having to update the mapping every time the schema changes.

In many application areas, the risk of having to update the mapping is either negligible or consciously taken into account, since data with real semantics is required. For these cases, an additional, manual mapping from the Relational.OWL representation to a target ontology would be appropriate. This may easily be done using a suitable query language. Please note, that all present relational database to Semantic Web approaches require the mapping to be updated, whenever the schema of the database changes, whereas our technique provides an automatic fallback for such situations.

Mapping within the Semantic Web: The complete mapping process from the relational database to RDF objects with real semantics is performed

using Semantic Web applications. As a result, two different mapping architectures are possible. The first and most reliable possibility is, that such mappings are processed by small wrapper applications providing the Semantic Web applications with the required target data. Taking place within the Semantic Web, the applications may nevertheless opt to create the mapping by themselves using their own built-in functionality.

Well-known Mapping Language(s): One of the main advantages of our approach is, that it does not require a new mapping language to be adopted, since it is completely based on current RDF/OWL-techniques. Contrary to approaches like [Biz03], Semantic Web application developers needing access to data actually stored in relational databases do not have to learn yet another mapping language, but are able to use their preferred RDF query language, as long as it fulfills the requirements mentioned in Section 3.3.1.

3.3.5 Classification

Mapping relational data, i.e. the Relational.OWL representation of a database to a target ontology can be regarded as part of the classical schema integration problem. Consequently, we want to assign our mapping approach to one of the two well known integration strategies *global-as-view* (GAV) or *local-as-view* (LAV) (cf. [Ull97, Hal01, Len02]). In GAV, for each component of the global schema, a view over the local sources is created. Consequently, as soon as one of the source schemas is altered, or a new source database is added, the mapping or even the actual global schema has to be revised. In contrast, LAV expresses each local source as a view over the global schema. The main advantage of LAV is, that whenever a source schema changes or is added, only the corresponding mappings have to be changed without affecting the global schema.

Dealing with a target ontology, which is rarely modified, used by an unknown amount of data sources, and not maintained by the local administrator, a GAV-based approach is not applicable. On the other hand, we do not use a LAV-based approach neither, since our sources are not described as a view over the global schema. Instead, we deal with multiple source databases, each mapping (part of) its schema to (parts of) the mainly static global target ontology on its own. This mapping approach hence resembles more a combination of GAV and LAV, like BGLaV, which is described in [XE04]. A detailed analysis, whether an approach using source-to-target mappings based on a predefined global ontology enables efficient query rewriting from the local database to the target ontology, as claimed by the authors, is left for future work.

3.3.6 Sample Mapping

In this section we present a sample relational data to RDF/OWL mapping using SPARQL as our chosen mapping language. Despite its novelty, SPARQL is already supported by the Jena Framework [Jen06].

Consider a Semantic Web application developer, who requires access to the data stored in the database introduced in Section 3.1. Since he assumes the database schema to be quite stable, he decides to create a mapping from the relational data model to Semantic Web objets based on the `vCard` ontology [Ian01]. A possible mapping query, which gives Semantic Web applications the possibility to access the data using its own built-in functionality and enables them to perform common reasoning operations is given in Figure 3.8.

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbinst:<http://www.dbs.cs.uni-duesseldorf.de/RDF/address_schema.owl#>
CONSTRUCT {_:v rdf:type vCard:ADR;
               vCard:Street ?street;
               vCard:Locality ?locality;
               vCard:Pcode ?pcode;
               vCard:Country ?country}
WHERE {{?a rdf:type dbinst:ADDRESS;
           dbinst:ADDRESS.ZIP ?pcode;
           dbinst:ADDRESS.STREET ?street;
           dbinst:ADDRESS.CITY ?locality;
           dbinst:ADDRESS.COUNTRYID ?x}.
       {?d rdf:type dbinst:COUNTRY;
           dbinst:COUNTRY.NAME ?country;
           dbinst:COUNTRY.COUNTRYID ?x}}
```

Figure 3.8: Sample mapping query

After specifying the prefix definitions for `vCard`, `rdf`, and `dbinst` in the `PREFIX` clause, the skeleton of the resulting RDF objects is defined in the `CONSTRUCT` part of the query. At first, a new blank node of type `vCard:ADR` is created. This object contains the attributes `vCard:Street`, `vCard:Locality`, `vCard:Pcode`, and `vCard:Country` and could easily be extended by further attributes either specified in the `vCard` ontology or in other RDF-Schema files. The values corresponding to the given attributes are specified by free variables, bound in the following `WHERE` clause.

The actual linkage to the original database is performed in the `WHERE` clause of the SPARQL query, i.e. each of the free variables specified in the `CONSTRUCT` clause is bound to a column of the original database. To be more precise, the attributes are bound to the data instances $\mathcal{I}$ of the RDF representation $\mathcal{ROWL}$ of the relational database. Please note, that the mapping specification is identical for a *virtual* RDF model like in RDQuery or a *static* data representation, e.g. with

the Relational.OWL application. Being stored in two different tables (`ADDRESS` and `COUNTRY`), the required data is joined using the `?x` variable (cf. Section 3.2.2).

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:vCard="http://www.w3.org/2001/vcard-rdf/3.0#" >
 <rdf:Description rdf:nodeID="A0">
     <vCard:Pcode>40225</vCard:Pcode>
     <rdf:type rdf:resource="http://www.w3.org/2001/vcard-rdf/3.0#ADR"/>
     <vCard:Street>Universitätsstr.  1</vCard:Street>
     <vCard:Locality>Düsseldorf</vCard:Locality>
     <vCard:Country>Deutschland</vCard:Country>
 </rdf:Description>
 <rdf:Description rdf:nodeID="A1">
     <vCard:Pcode>31006</vCard:Pcode>
     <rdf:type rdf:resource="http://www.w3.org/2001/vcard-rdf/3.0#ADR"/>
     <vCard:Street>Campus de Arrosadia</vCard:Street>
     <vCard:Locality>Pamplona</vCard:Locality>
     <vCard:Country>España</vCard:Country>
 </rdf:Description>
</rdf:RDF>
```

Figure 3.9: Sample mapping result

Having created a suitable mapping from the Relational.OWL representation of the database to the target `vCard` ontology, the resulting data can be processed by most Semantic Web application as usual. In Figure 3.9, a sample result set of the mapping query provided in Figure 3.8 is given.

3.3.7 Evaluation

We have evaluated the performance of our relational database to Semantic Web approach using RDQuery (cf. Section 5.2). It is a wrapper system, which enables Semantic Web applications to access and query data actually stored in relational databases using their own built-in functionality. RDQuery automatically translates SPARQL and RDQL queries into SQL and is hence able to perform the relational to semantic mapping in one step. Providing an adequate mapping query, the query is translated into SQL, the underlying relational database is queried, and the results are returned in the required format. RDQuery is thereby able to recognize the basic operations of the relational algebra within the SPARQL query (cf. Section 3.2.2) and to translate them into SQL. Hence, most of the workload, including join and projection operations, is not processed directly by RDQuery, but passed to the underlying database with the generated SQL query.

Following the example shown in Section 3.3.6, we have created 18 different queries which map relational data to the vCard ontology (cf. `http://www.w3.org/TR/vcard-rdf`). We have categorized these queries into three different classes, depending on their complexity referring to the relational algebra (i.e. `selection`, `projection`, and `join`). Each of the categories contains six of the

	Execution time [s]	
SPARQL Query	Query Translation	Mapping Process
Selection	0.020	0.050
Projection	0.018	0.052
Join	0.023	0.067

Table 3.3: Average execution time for a SPARQL mapping

queries. We first measured the time required by RDQuery to translate the queries into SQL and then the time passed for the complete mapping process, including query translation, query execution via JDBC using a MySQL database, and the data translation back into RDF. The database, the queries were tested on, is based on the `northwind` database and contains eight tables with a total of about 3000 tuples. Unlike the first measurement, the second depends on various factors, like network or database performance, which can hardly be influenced by RDQuery. In Table 3.3, the average execution time of the query translations and mapping processes for each mapping category is given.

The performance results show, that the execution time of both, the query translations and the complete mapping process are barely measurable, lying most of them far below 100 milliseconds. Even the more complex join operations were translated and executed at an average of 67 milliseconds. Consequently, our mapping relational data to Semantic Web approach enables applications to access legacy data stored in relational databases in real-time, as if that data would actually be part of the Semantic Web.

3.4 Related Work

Since the raise of XML in the late 1990s and early 2000s a large number of different exchange formats for relational database systems have been developed based on XML (e.g. Torque as part of the Apache DB Project [The04]). Actually, each vendor of (object)relational database systems has tried to establish its own XML representation, like ORACLE's `XMLElement` function or IBM's `Rec2XML` scalar function, already mentioned above. Since these different dialects can easily be converted using XSLT, this babylonic chaos of different representation languages is broadly accepted. Nevertheless, the transformation rules between the different dialects have to be created manually.

After Berners-Lee et al. had expressed their vision of the next generation Web, a Semantic Web in [BLHL01], the community started to seriously adopt the idea of semantic reasoning within the World Wide Web. This includes also some ideas on how to extract data from (relational) database systems, whereof [BL98a] can

be seen as an early forerunner. Nevertheless, this mapping of relational data to RDF is rather rudimental, e.g. lacking a concrete schema representation.

In [Mel99], Melnik introduces a working draft for an algebraic definition of RDF models. Unfortunately, this proposal does not include the operations required for querying the RDF graph. A more sophisticated approach is presented by Frasincar et al. [FHVB04]. They introduce a complete algebra for RDF, inspired by the relational algebra. Consequentially, the authors do not only introduce the corresponding algebraic data model, but also adequate operators like the projection, the Cartesian product, or the selection. Nevertheless, the authors did neither make a direct comparison to the relation algebra, nor explain how to bring the relational and semantic worlds together.

Recently, more efforts arose in bringing together relational databases and the Semantic Web. Anyhow, most of these approaches do not use relational databases as a data source, but to store RDF triples in tailored tables, exploting the improved query performance of current relational databases (e.g. [KCPA01], [PH03], or [HS05]). The main drawback of such approaches is, that the corresponding data has to be available in RDF, i.e. their aim is not to convert legacy data into a Semantic Web representation, but to give applications fast access to RDF triples.

Karvounarakis et al. present for example RQL [KCPA01], an advanced declarative query language for RDF schemas and descriptions. To achieve the best possible query results, the authors store the RDF data in a relational database, where the actual query is performed. Therefore, they make an adequate mapping of all RDF triples to the relational data model. Since this mapping is specific to the needs of querying *normal* RDF triples, stored in a special way in a relational databases, it is hard to apply it to our needs. In fact, we would have to transform data stored in relational databases to RDF/OWL using Relational.OWL, just to be stored once again in a relational database for accessing it.

Nevertheless, there are also some techniques, trying to map legacy relational databases to the Semantic Web. Bizer [Biz03, BS04] for example, introduces a mapping from relational databases to RDF. Unlike our approach which is based on existing query languages, this method requires a specific mapping language, which, although it is based on RDF, still has to be learned and adopted by the corresponding developers.

Semantic integration of corporate information resources is the main topic in [BJY+02], where Barrett et al. use RDF as a standardized communication language between multiple components. In their approach, relational data is linked to an ontology, which itself is used as a neutral interchange format. Hence, the aim of the authors is not to represent relational data in a semantically processable way, but to relate the concepts stored in the database with those of an RDF ontology.

An et al. outline in [ABM05] a further approach from tables to ontologies. Unlike our technique, this approach maps database schemas directly into ontological concepts, assuming that the required database was designed following several ER

design principles, e.g. the database is normalized and contains meaningful table or column names.

Petrini and Risch introduce in [PR04] their technique to query relational databases using RDF query languages, which is closely related to our approach. Nevertheless, it has some drawbacks. The mapping from relational tables to the Semantic Web is defined within a custom made mapping table, where columns or tables are related to objects or attribute values. As a result, the mappings between both worlds are always one-to-one. Our mapping technique is completely based on the Semantic Web and allows the mappings to be as complex as a query language can be, i.e. we would even be able to use aggregations, if they are supported by the query language used.

Haase et al. provide in [HBEV04] a survey describing different RDF query languages. The paper starts with the promising challenge to examine, whether the query languages are relational complete or not. Since the paper examines six different query languages it analyzes the languages superficially, omitting the argumentation for the conclusions. In fact, the authors seem to have underestimated the expressiveness of RDQL. Unfortunately, the paper does not include SPARQL, since it was released almost at the same time. A quite complete overview of current query languages for XML and the Semantic Web is presented by Bailey et al. in [BBFS05].

3.5 Discussion and Future Work

We have seen in this chapter how to represent the schema of a relational database using our Relational.OWL ontology. The semantic representation of the schema itself can then be interpreted as a novel ontology, i.e. a schema ontology. Based on this tailored schema ontology, the data stored in that specific database can be represented within the Semantic Web.

The next step was to analyze whether such a relational data and schema representation could potentially replace existing interfaces for the access of relational data out of the Semantic Web. The advantage of such an approach is obvious: All Semantic Web applications could query data stored in relational databases with their own built-in query languages, without having to convert this data in a manual and time-consuming process.

A direct comparison of this approach with SQL, the commonly used interface to relational databases, would be unfair and easily get out of hand, since SQL has evolved during the last decades and most semantic query languages are rather rudimental. Hence, we have decided to check if it is possible to express the basic operations of the relational algebra with one of the upcoming query languages for the Semantic Web.

During our analysis we observed, that the fundamental drawback of RDQL is the absence of closeness, i.e. the result set of a query is not a valid RDF/OWL

expression any more. We hence have decided to recheck the basic operations using the closed query language SPARQL.

Despite some differences to the relational algebra, e.g. in the Cartesian product, we managed to simulate the basic operations $\{\sigma, \pi, \cup, -, \times\}$ of the relational algebra. Additionally we showed, that join operations, which will certainly be part of most mapping operations between the relational and the semantic worlds, can easily be deduced from our Cartesian product simulation — just as it is done within the relational algebra. Since we were successfully able to simulate more than the basic relational operations, we have shown, that the combination of Relational.OWL and SPARQL are relational complete.

After being able to query the Relational.OWL representation of a relational database, we have described how to map data from relational databases into a real RDF representation using a Semantic Web query language. To use such query languages for a mapping purpose, three main requirements have to be met. First, the relational database (i.e. its schema and data components) has to be described using the Relational.OWL ontology. This automatic semantic representation of the relational database can then be queried using any RDF query language. If the adopted query language is closed, the resulting RDF graph can be specified to match the target ontology, the original database shall be mapped to.

The approach presented in this chapter is based on mapping the Relational.OWL representation of relational databases manually into Semantic Web objects with real semantics. We are thus planning to analyze, whether existing (semi-)automatic schema and ontology matching approaches (cf. [SSC05, DMDH04, RB01]) could provide reasonable results in matching an existing relational schema to a target ontology, i.e. to find correspondences between the schema of the database and the target ontology.

The expressiveness within the mapping process depends directly from the query language used, i.e. a more complex mapping cannot be stated with an elementary query language. For instance, we showed that all the basic operations of the relational algebra can be expressed with SPARQL. Nevertheless, its has some considerable limitations, since it does not support aggregations or nested queries. A further restriction concerns data manipulation or data updates, which is still not supported by most RDF query languages. We are currently analyzing, whether SPARQL could be extended to support such operations for enabling Semantic Web applications to manipulate the data actually stored in the relational database.

Chapter 4

A Novel URI for Databases

In chapter 3 we have introduced our technique to map data stored in relational databases into the Semantic Web. The mapping process consists of two main steps, one automatic data and schema extraction using the Relational.OWL ontology and a subsequent mapping to a specific target ontology. During the first step, we represent the data and schema items of the database like they were in the database, i.e. the data tuples are instances of their own schema (ontology). Nevertheless, due to the lack of a suitable URI schema, we loose one important information: the exact location of the originating database server and its corresponding data and schema components. As a result, it becomes impossible to backtrack the data to its original data source or to identify two data tuples as actually being the same.

In this chapter we suggest the novel URI scheme `db` for identifying not only databases, but also their schema and data components like tables or columns. One of the features of this scheme is that it may not only be used for relational database systems, but for virtually any type of database or data source. We therefore have combined the advantages of both global uniqueness of URIs and the high flexibility of knowledge representation with RDF as part of the Semantic Web. With this novel identifier we are now able to enhance every data record exchanged between databases with metadata: an exact and identifying location of that data in the data source. As a result not only the system administrator is able to backtrack the data to its exact position in the data source but also the database system itself. The approach presented in this chapter is largely based on work published in [PC03].

4.1 Motivation

Nowadays data is not exchanged and integrated manually anymore, but semiautomatically. Data exchanges between data sources run on their own after being set up once by a database engineer. Afterwards he only has to interfere if an error

occurs or changes have to be done. In the former case the source of the data which produced the error has to be backtracked, so the error can be reconstructed and solved. This seems quite easy in an environment with two databases involved, but in more realistic environments we usually have quite more databases engaged in a data exchange. Since the data exchange formats and the data exchanged may only differ marginally, assigning the data to its corresponding sources may be quite complicated or even impossible.

Following the vision of completely autonomous databases exchanging not only their data, but also their metadata [BGK+02, HIMT03], we will soon face a similar problem. In these cases, not only the data, but also the schema components have to be backtracked exactly to their sources. If the data source cannot be identified unambiguously out of the data exchange process, special identifiers for the data sources are needed.

Data sources do not only have to be identified in ongoing data exchanges, but also for later analysis. The information where a data record came from could be crucial especially if errors do not appear immediately. If, in the meantime, the same data has been exchanged with multiple other data peers, the actual source of that item cannot be traced any more.

In many cases, internally assigned identifiers for each specific data source would be enough. However if we start having autonomously and automatically acting databases exchanging data and metadata, this internal identifiers will not suffice any more. We can illustrate this with a small example.

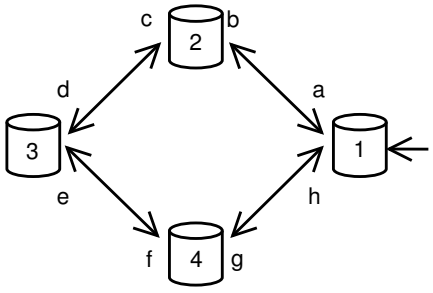

Figure 4.1: Multi-Peer-to-Multi-Peer data exchange

A data exchange system consisting of four databases (cf. Figure 4.1) where each database synchronizes its data with two of the other databases. Each database propagates all data changes to both exchange partners, for which it has assigned an internal name, i.e. database 2 (db2) knows database 1 (db1) by the name *a*, db1 knows db2 by *b*, etc.

A data update on db1 is propagated to db2 and database 4 (db4). The first recognizes the data as coming from database *a*, for db4 the data comes from *h*. After it has been imported into the respective databases, the data records have to be propagated again. For this reason database 3 (db3) will get data from db2

and db4. Even if there was an identifier of the original source attached to that data, db3 would never be able to recognize the data as coming from one and the same source. Since the source database has two different names (a and h), db3 is not able to recognize these two sources as being the same.

Apparently it makes sense to have a global identifier for databases. This should not only identify the database itself, but also its components. The purpose of this chapter is to propose such a global identifier.

4.2 Challenges Designing an Identifier

The first step in designing an identifier for databases is to analyze existing standards, whether they may be suited accordingly. Obviously we first think of the *Abstract Syntax Notation One* (ASN.1) [Int02] or the *International Code Designator* (ICD) [Int98], but they do not fulfil our needs. ASN.1, a standard for the identification of all kinds of objects, has three main disadvantages. First, every company or institution interested in the identification of their database objects would have to register a subset of the ASN.1 number space, which would entail additional costs. Second, the evolution of semi-structured data to XML and the high popularity of domain names instead of IP addresses have shown that a data or address–exchange has to be done in a human readable and understandable manner. An approach based on ASN.1 with its predominating representation *Object Identifiers* could be understood only with the aid of manually maintained mapping–tables. And finally there is no way of syntax validation included in the ASN.1 standard [Mea00]. Especially the possibility to validate the identifiers would give us an important instrument en route to an automated error-backtracking within autonomous and automatic data exchange environments.

Since ICD is a subset of ASN.1, the disadvantages mentioned above also apply. In addition, the number of possible enterprisers with assigned ICD numbers is limited to 9000 [Int98], a tremendously undersized amount for identifying databases of potentially each company or institution in the world.

Apart from the arguments mentioned above, ASN.1 and ICD identifiers persist only of a global identification number similar to URNs [Moa97], but do not include a locator. Thus databases, trying to backtrack a data's source, would not be able to locate it, nor to contact it. For this reason, and especially taking autonomously acting databases into account, the location of a data source is an indispensable information and should hence be included in the identifier.

The most important feature of the identification mechanism must be its flexibility. It has to suit to relational and object-oriented databases as well as to directories, legacy systems, or database types not yet developed. Additionally, we have to handle with heterogeneity within the different database types themselves [VJBCS97, KCGS95]. A relational database server from vendor a may consist of different databases in which every database user has an assigned table

space. Another relational database (even from the same vendor) may skip the database-user hierarchical level, so that one big table space is shared by all users. This sort of heterogeneity has to be considered as well.

Of course we could propose to introduce a new registration authority for worldwide unique database identifiers, but this center needs to be established first, the service would certainly not be free of charge, and it would have to be accepted by the community.

Concluding, we require a flexible identifier for databases and their components, without having to establish a registration authority. For thus purpose, we have decided to combine normal URIs [BLFM98] with RDF [Las97], the principle technology of the Semantic Web [BLHL01, Dum01]. The combination of both URIs and RDF enables us to create a flexible identifier with a virtually exhaustless address space, which is human readable and at the same time machine understandable.

4.3 Model

As mentioned above, we propose a novel model for identifying databases and their components combining the advantages of both Uniform Resource Identifiers (URI) and the Semantic Web.

A URI is a "compact string of characters for identifying an abstract or physical resource" [BLFM98]. It is predestinated for creating a global identifier for databases and their components. Presently, there are several standardized URI schemes, for instance those for telephone numbers, email addresses, or host specific file names [Int03e], but the best–known and most used URI scheme is `http`, which stands for the Hypertext Transfer Protocol [FGM+99]. All URI schemes were introduced to create a global identifier for specific resources, which is also our aim. Since there is still no appropriate URI scheme for databases, we introduce the URI scheme `db` for databases. It identifies databases and their components, no matter of what type the database is and how that database is composed.

Each database involved in a data exchange can be reached using its unique IP address or domain name. We are thus able to use this address for the identification of the database server itself as the first part of our novel identifier. Additionally, we want to identify its components, for example its tables or columns which can usually be done through a kind of hierarchy. Since we cannot give a global hierarchy for all sorts of databases, the URI scheme itself has to be hierarchy independent. We have decided to use the Resource Description Framework (RDF) [Las97] of the Semantic Web for identifying single database components. The advantage of RDF is definitely its flexibility. Referencing to an adequate ontology, we are able to create arbitrary chains of attribute/value pairs for the unambiguous identification of the required components. Using an ontology like the Relational.OWL ontology (cf. Section 3.1), a database column named `WorkerPK` could

be identified by `Database=admin and Table=Worker and Column=WorkerPK` or just by `Column=WorkerPK`, depending on the uniqueness of this specific column name. Besides the flexibility achieved through RDF, this Semantic Web technology gives the attribute/value chain a real meaning, i.e. we are able to represent actual knowledge. Hence, a remote database would know, that `dbs:Table` represents the construct *relation* known from the relational model.

Summarizing, we are able to identify a database server using its unique IP address or Domain name and its components using an RDF-based attribute/value chain. Unfortunately, there are still some challenges to be considered:

1. how to represent the RDF-Syntax [LS99] within the URI,
2. how to use internationalization inside the URI,
3. how to specify the corresponding namespaces,
4. how to address database servers located in private networks,
5. how to incorporate temporal aspects, and
6. how to deal with databases changing their IP addresses constantly or having a private one.

Challenges one, two, and three are solved below, the fourth, fifth, and sixth are subject to further research.

The most evident syntax for a database URI composed of an IP address and several RDF attribute/value pairs would certainly be `db://ipaddress/prefix:attribute=value&prefix:attribute=value`, where `ipaddress` specifies the location of a database server, `prefix` is a namespace pointing to the ontology used as a vocabulary of database components, and `value` is the concrete name of the corresponding database component.

Unfortunately this syntax is not allowed, since the colon is a reserved character for specifying the port [BLFM98] and thus must not be used elsewhere within a URI. Using the ASCII representation `%3A` of the colon could also lead to problems, since the URI will most likely be used in XML documents and the `%` character specifies a parameter-entity within the DTD of an XML document [BPSM04]. For such cases, we have to provide a further alternative representation of the colon, i.e. `&#58;`. Consequently, an attribute/value pair specifying the location of a database component looks like `prefix%3Aattribute=value` or like `prefix&#58;attribute=value`.

Using the URIs in XML documents leads to a further challenge, since the ampersand (`&`) specifies the beginning of an entity reference or a special character, like `&#58;` for the colon. We hence additionally allow the usage of a semicolon (`;`) as an alternative for separating the attribute/value pairs form each other.

A further problem arises with non-standard ASCII characters in databases. We cannot assume all the names of tables or columns be restricted to the limited amount of ASCII characters, especially if we have to handle with data exchanges crossing different language regions. In the RFC for the URI Generic

Syntax [BLFM98] this restriction is virtually made, since it is not mentioned how to deal with non-standard ASCII characters. However, since the URI will always appear within a data exchange where a corresponding character encoding has already been arranged, we define the encoding of the URI to be the same as that of the document it is contained in. Was the encoding of the data exchange defined in the XML header as `ISO-8859-1`, we would be able to have `prefix%3Aattribute=küche` as part of a valid URI. With the encoding `ISO-8859-5` a valid URI could contain `prefix%3Aattribute=кухня`. A corresponding example could be given for virtually any encoding.

Besides the encoding, the namespace, i.e. the prefix representing the ontology, has to be defined before a corresponding attribute can be used within the URI. Defining it within the URI itself would lead to a large overhead, especially if the same namespace is used within several URIs. We hence assume that all the namespaces have already been defined within the data exchange document. We are hence only allowed to use a prefix `dbs` if it was defined as being the representative for a correponding ontology, e.g. `http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#`.

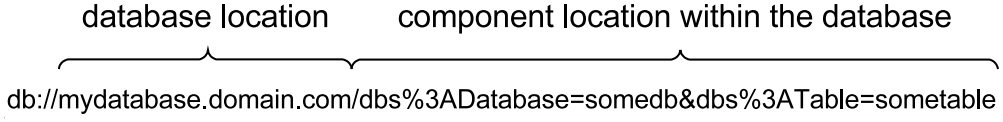

Figure 4.2: The `db` URI scheme

To sum things up, the novel URI scheme `db` presented in this chapter consists of two main parts (cf. Figure 4.2). The first part represents the location of the database server, i.e. its IP address or domain name. The second part specifies the location of the component in the database using a chain of RDF attribute/value pairs. The delimiter between the attribute/value pairs is either an ampersand (`&`) or a semicolon (`;`). Whereas the prefix and the attribute names are either separated by `%3A` or by `&#58;`, both representing the colon (`:`).

4.4 Example

After having defined our URI scheme for databases we will now give a short example. Given the fact, that such a novel URI could improve both, XML and RDF-based data and schema representation formats of relational databases, we describe both scenarios. First we show how to embed the URI into a basic XML data exchange and then how to include this URI into the Relational.OWL representation of a relational database (cf. Section 3.1). The description of both techniques are again based on the database introduced in Section 3.1, which contains the following two relations:

`Address(AddressID, Street, ZIP, City, CountryID)` and `Country(CountryID, Name)`.

4.4.1 XML Data Exchange

Representing the data stored in the relational database using a simple XML-based exchange format could lead to a file as given in Figure 4.3. Since there is no constraint enforcing the tags to be named after the table or column names, the tags may not give information about the actual storage location in the database.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<mydatabase>
  <Residence>
    <PK>3248</PK>
    <Str>Königsallee 21</Str>
    <ZIP>40212</ZIP>
    <City>Düsseldorf</City>
    <Country>32</Country>
  </Residence>
  <Country>
    <ID>32</ID>
    <Name>Deutschland</Name>
  </Country>
</mydatabase>
```

Figure 4.3: XML data exchange document

Using our novel URI scheme `db` we are now able to incorporate additional metadata into that exchange file. Adding a source-specific URI to each data record, we are now able to backtrack a data item to the exact storage position in the original data source. Adding this information to the data exchange file from Figure 4.3 results in a exchange file as given in Figure 4.4.

After specifying the XML version and the corresponding character encoding, we have to define the entities used within the exchange document. Since all of the URIs used within the document are partly identical, we define three entities, which can be used further on within the document. We are hence able to reduce the amount of data overhead to a reasonable amount. The entity `mydb` corresponds to the location of the actual database and the entities `address` and `country` correspondingly to its tables `Address` and `Country`. Besides the entity definition, there are two additional differences to the basic data exchange file in Figure 4.3: a namespace definition and the source specification for each data item. The namespace `dbs` pointing to an ontology for relational databases is defined

in the outmost element and may hence be used within the complete XML document. This is the namespace required for our URI. The attribute `src`, which was added to the elements, specifies the source where the data actually came from, i.e. the data between the opening and closing `mydatabase` tags came from the database named `admin`, stored in the server with the domain name `foo.de`. The source specifications for the remaining tags are made accordingly.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE mydatabase [
 <!ENTITY mydb "db://foo.de/dbs&#58;Database=admin">
 <!ENTITY address "&mydb;;dbs&#58;Table=Address">
 <!ENTITY country "&mydb;;dbs&#58;Table=Country">
 ]>

<mydatabase src="&mydb;" xmlns:dbs=
      "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#">
 <Residence src="&address;">
  <PK src="&address;;dbs&#58;Column=AddressID">3248</PK>
  <Str src="&address;;dbs&#58;Column=Street">Königsallee 21</Str>
  <ZIP src="&address;;dbs&#58;Column=ZIP">40212</ZIP>
  <City src="&address;;dbs&#58;Column=City">Düsseldorf</City>
  <Country src="&address;;dbs&#58;Column=CountryID">32</Country>
 </Residence>
 <Country src="&country;">
  <ID src="&country;;dbs&#58;Column=CountryID">32</ID>
  <Name src="&country;;dbs&#58;Column=Name">Deutschland</Name>
 </Country>
</mydatabase>
```

Figure 4.4: XML data exchange document with our novel URI

4.4.2 Relational.OWL Representation of a Database

Representing the data and schema of a relational database based on the Relational.OWL ontology (cf. Section 3.1) means to incorporate data originated from relational databases into the Semantic Web. Although we are able to represent the basic semantics of the relational schema in a machine readable and understandable way, one essential information is not included: the location of the original data source.

As we have seen in Section 4.1, the information about the origin of a data item can be very important, especially in scenarios where data is being exchanged among multiple partners. Dealing with data on the Semantic Web, particularly if inference rules shall be applied, the unambiguous identification of a data item using URIs becomes vital.

Currently, it is hardly possible to identify two data items as being the same

information, i.e. representing the same data tuple in a source database, since present implementations of Relational.OWL (cf. Section 5.1) use local identifiers without a global uniqueness. In fact, for most scenarios, providing relational data in a semantically processable way is enough, since Semantic Web applications are able to access data using their own built-in functionality, which was previously out of their reach. Nevertheless, for some advanced reasoning tasks, a unique URI identifying the data item stored in the data source would support meeting the challenge. In this section we sketch up some ideas on how to include our novel URI into the Relational.OWL representation of a relational database for supporting such Semantic Web applications in their reasoning tasks.

Adding the source specific URI to the data exchanged using a basic XML document (cf. Section 4.4.1) was rather straightforward: we added the corresponding URI pointing to the original storage location to each individual data item. The decision where to add the novel URI within the Relational.OWL representation of a database is not that trivial, since we have to deal with two different representation types, a data and a schema representation (cf. Section 3.1).

Including the URI into the data representation according to the XML data exchange in Section 4.4.1 would mean to dilute the strict separation between the schema and the data representation, since we would include metadata into a pure data representation. The adequate position for such a URI is thus the schema ontology of the database.

Nevertheless, adding the corresponding URIs to the schema representation does not mean to loose this information in the actual data representation. Since the data items are instances of the schema classes defined in the schema ontology, this information is implicitly available within the corresponding data representation.

There are basically two different approaches on how to include the novel URI scheme into the schema representation of a relational database. On the one hand, the Relational.OWL ontology could be extended to include an additional attribute `src` for specifying the URI of the corresponding schema components. On the other hand, it would be possible to replace the IDs of the schema component representatives using our novel URIs.

To add an additional attribute to the Relational.OWL ontology would probably be the easiest and quickest way to incorporate the URIs into the schema file. Nevertheless, this option would result in an imprecise model of the database schema, since the URI is not a property of a schema component, but it represents the schema component itself. It would hence be more precise to replace the current IDs of the schema component representatives with their source-specific URI.

However, even this approach has one big disadvantage, since many Semantic Web applications assume the URI of an object to be a real URL, i.e. to point to the location, where the corresponding Semantic Web object is defined and where to acquire additional information about this object. In our case, the URI does

not point to such a location, but represents the logical location of a component in the database server. Consequently, this approach could lead to difficulties in the reasoning process, especially if reasoning engines try to access the resource implied by the URI.

```
<...>
 <owl:DatatypeProperty rdf:ID="ADDRESS.ZIP">
  <rdf:type rdf:resource="&dbs;Column"/>
  <rdfs:domain rdf:resource="#ADDRESS"/>
  <rdfs:range rdf:resource="&xsd;string"/>
  <dbs:length>8</dbs:length>
  <dbs:src>&address;;dbs&#58;Column=ZIP</dbs:src>
 </owl:DatatypeProperty>

 <owl:DatatypeProperty rdf:about=
                       "&address;;dbs&#58;Column=COUNTRYID">
  <rdf:type rdf:resource="&dbs;Column"/>
  <rdfs:domain rdf:resource="#ADDRESS"/>
  <dbs:references
         rdf:resource="&country;;dbs&#58;Column=COUNTRYID"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
 </owl:DatatypeProperty>
</ ...>
```

Figure 4.5: Relational.OWL-based schema file using our novel URI

Since it is out of the scope of this work to decide, which is the best approach to include the URI into the Relational.OWL representation of a database, we have included both possibilities into a small example. The sample schema representation in Figure 4.5, which is based on the schema representation of Figure 3.3 on page 38, contains the Relational.OWL representation of two columns. The first column was enhanced with an additional property named `dbs:src`, which currently is not part of the Relational.OWL ontology, but may easily be added. In the other column we have implemented the second approach and replaced the `rdf:ID` attribute by an `rdf:about` attribute containing the corresponding URI. Please note, that the `dbs` namespace and the `&address;` and `&country;` entities used for clarity reasons correspond to the entities already introduced in Figure 4.4.

4.5 Related Work

As far as we are aware of, there is only one alternative approach to our URI scheme for relational databases. In [BL98a], Berners-Lee compares the relational and the Semantic Web models. Among his ideas on how to map relational data into the Semantic Web is a scheme for specifying corresponding URIs. The structure of our URI is similar to that presented in his work, i.e. both approches consist of two parts: the location of the database server and the component location within the actual server. Nevertheless, there are some major differences in specifying the location of a specific database component. The approach presented by Berners-Lee does not use knowledge representation techniques to specify the location of a component, it exclusively uses a hierarchy of component names, i.e. the URI `http://www.acme.com/mycat/` could either represent a database catalog, a database instance, or a user schema, depending on the hierarchy implemented in the concrete database management system. As a consequence, a remote database system is not able to interpret such a URI unambiguously, since the encoded information has lost its semantic significance completely.

4.6 Discussion and Future Work

In this chapter we have suggested a novel URI scheme for identifying and locating not only databases themselves, but also their schema and data components. This scheme guarantees a global uniqueness of the identified components and remains as flexible as RDF is. Herewith we are not only able to identify relational or object-oriented databases but also legacy or file systems.

The `db` scheme consists of two parts. The first part is the locator of the database (e.g. its IP address) and the second part, realized with RDF, points to a location inside the database system. Using RDF means to not only write attribute/value pairs, but to represent knowledge about the structure of the database. Herewith, potential data and schema exchange partners, being humans or computers, are instantly able to read and understand the data received from remote data sources.

Besides the challenges mentioned in previous sections concerning Relational.OWL and our novel URI scheme, there are some additional items to be solved. The most important task is to incorporate a temporal aspect [SPZ98] to the URI scheme. Herewith we would be able to identify a data item indefinitely, no matter how often it was changed on the actual data source. A reasonable solution could be to introduce a mandatory attribute giving the time when the data record was created, e.g. `foo:recordCreationTime=1057159782` using the UNIX time or `foo:recordCreationTime=2003-07-02T152942GMT` using one of the numerous ISO 8601 [Int88] formats.

A further challenging problem concerns databases with changing or not reach-

able IP addresses, e.g. mobile databases or those in private networks. They either change frequently their address, or do not have an unambiguous one. This could probably be evaded introducing artificial addresses or names. Please note, that this fact affects only the first part of the URI scheme and not our method of identifying components within the database.

Needing several attribute/value pairs for specifying a component within a database usually leads to lengthy URIs, which significantly increase the data overhead within a data exchange or representation file. We hence have to consider some further steps for abbreviating the second part of the identifier. A step would be to use XML entities for common parts of the URIs, like we have done in both examples. A further step could be to use a standard namespace, i.e. if only a basic ontology is required for describing the location on a database, the namespace specification would not be required any more.

Chapter 5

Applications

Having introduced our Relational.OWL approach to extract the schema and data items of a relational database and to translate it into a semantically rich format, we now present several applications, in which the Semantic Web representation of relational databases play a decisive role. We show with these applications, that the approaches presented and the results concluded in this thesis can directly be implemented and have an important impact on different application areas.

The presentation of the applications is organized as follows: In Section 5.1, we present two different implementations of Relational.OWL, which both are able to extract the data and schema components of a relational database and to transform them into their Relational.OWL representation. Thereupon, we show in Section 5.2 RDQuery, an application which applies the results achieved in Section 3.2 and automatically translate RDQL and SPARQL queries into SQL, enabling Semantic Web applications to query relational databases on the fly using their own built-in functionality. Finally, we present DÍGAME in Section 5.3, a P2P database architecture, which achieves its actual flexibility to include volatile peers through a Relational.OWL-based data exchange format.

5.1 Relational.OWL Implementations

In this section we show in practice how to translate the schema of a relational database into a schema ontology and then how to extract the corresponding data instances and convert them into RDF. We therefore present two applications, which are both able to access relational databases and automatically translate their schema into a schema ontology and the corresponding data items into instances of this newly created ontology. The main drawback of both implementations presented in this section is the up-to-dateness of the extracted data. The result of both applications is a Relational.OWL representation of the database, which states the database at the translation time, i.e. if a database is altered afterwards, the Semantic Web representation becomes out-dated. In order to achieve

a synchronized semantic representation of the database, the complete schema and/or data extraction process has to be repeated whenever a modification occurs. Nevertheless, having a hard-copy of the relational data in its Semantic Web representation, any Semantic Web application is able to process this data, even if the underlying database has crashed or is no longer available.

We first present the Relational.OWL application, a Java-based database-independent framework, which accesses a relational database using JDBC and translates its corresponding data and schema components into their Relational.OWL representation. Thereupon we show how to implement Relational.OWL directly within a specific database management system, without having to use a programming language like Java, we only use XML technologies like XSLT or XQuery.

5.1.1 Relational.OWL Application

The Rleational.OWL application is a Java framework, which extracts the data and schema components of a relational database and automatically converts them into their Relational.OWL representation, i.e. it makes the corresponding components accessible to the broad majority of Semantic Web application, as long as they are able to process RDF and OWL. The Relational.OWL application is published under the GNU GPL and can be downloaded from `http://sourceforge.net/projects/relational-owl/`. This section is based on the documentation provided at that site.

5.1.1.1 Introduction

Despite the facility to gain new knowledge through reasoning mechanisms, most Semantic Web applications are bounded by the little amount of available data in a processable way, since most of the data is still stored in relational databases. We hence need applications, which access relational databases and provide Semantic Web applications with an understandable representation of that data.

The Relational.OWL application, implemented in Java, is such a framework, offering Semantic Web application the possibility to retrieve information actually stored in relational databases using their own built-in functionality. It therefore connects to potentially any relational database using the *Java DataBase Connectivity* (JDBC) [JDB06], converts its relational schema into a schema ontology and represents the data items correspondingly. Although this data extraction is actually based on the schema ontology created during the schema extraction process, both tasks can be performed separately, e.g. for updating the actual data representation, whenever a data modification occurs in the data source.

The usage of JDBC for accessing the database, enables the application to remain independent from the underlying database management system, i.e. it basically does not make any difference, whether the database management sys-

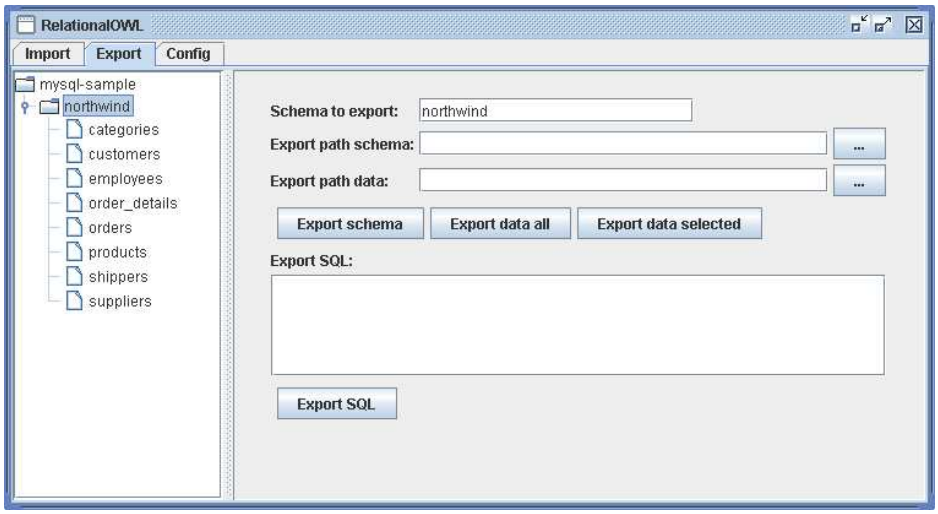

Figure 5.1: The Export tab of the Relational.OWL application

tem is from vender A or B, as long as a JDBC-driver is provided. Nevertheless, the JDBC implementations or SQL dialects of most database vendors have some differences, which makes it necessary to slightly adapt the application to the concrete database management system. Currently, the Relational.OWL application supports two different database management systems, the IBM DB2 Universal Database and MySQL, but it may easily be extended to support additional systems.

5.1.1.2 Usage

Depending on the required functionality, the user may opt to use the graphical user interface (GUI) of the Relational.OWL application or to embed the framework into an (existing) application using its API for accessing the Relational.OWL representation of a database. In both cases, the actual data transformation is performed automatically, without any human intervention.

GUI: As soon as the main class `de.hhu.cs.dbs.RelationalOWL` of the Relational.OWL application is started, the GUI window appears. The window consists of three tabs: *Config*, *Import*, and *Export* (cf. Figure 5.1), giving the user a quick overview of the functionality. All the properties required to connect to a database, including the class name of the JDBC-driver, the URL of the database, or the database credentials, have to be specified or loaded from a corresponding property file within the Config tab. After the *Reload* button has been pressed and the application is able to connect to the specified database successfully, it is ready to perform an import or export task. Depending on the desired functionality, the user may either switch to the Import or Export tabs. Besides the

export of a complete database, Relational.OWL is capable to export data based on an SQL statement, which can be specified in the Export tab. The remaining features of the GUI can be used intuitively.

Schema Export: As already mentioned, the user is not forced to use the GUI to benefit from the capabilities of the Relational.OWL application. Instead, he may opt to include its functionality into his own application using the API of the framework, which is shown for a sample schema export in the code snippet of Listing 5.1. Before the schema extraction can be performed, the database credentials have to be passed to the framework, which in turn tries to establish a connection to the database. Thereupon, the actual schema transformation is performed in a new thread, giving the calling application the ability to perform further operations, whilst the schema is translated. Finally, an XML representation of the RDF model just created is printed using the standard output stream `System.out`.

```
DbManager dbManager =
   DatabaseManagerFactory.getDbManagerInstance(connInformation});
ExportSchemaTask taskES = new
   ExportSchemaTask(dbManager.getConnection(), driver, database);
taskES.go();
while (!taskES.isDone()){
   Thread.sleep(100);
}
OntModel schema =
   ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM,null);
schema = taskES.getSchemaOntology();
schema.setNsPrefix("dbs",
      "http://www.dbs.cs.uni-duesseldorf.de/RDF/relational.owl#");
RDFWriter utf8Writer = schema.getWriter("RDF/XML-ABBREV");
utf8Writer.setProperty("allowBadURIs","true");
utf8Writer.setProperty("relativeURIs","same-document,relative");
utf8Writer.write(schema, System.out, "");
```

Listing 5.1: Schema export

Data Export: After the schema has been exported, the data instances can be processed. Although a schema file is not strictly required for the data extraction, it is advisable to specify a corresponding schema ontology, otherwise the data instances are not linked to their corresponding ontology and a subsequent reasoning task by a Semantic Web application may become impossible to be performed. The code snippet in Listing 5.2 shows how the data items of the corresponding database can be transformed into their Relational.OWL represen-

tation. Again, after the connection to the database has been established, the data instances are extracted and a corresponding model is created. After the assigned thread has performed its translation task, the data may be printed analogously to the schema model in Listing 5.1.

```
DbManager dbManager =
   DatabaseManagerFactory.getDbManagerInstance(connInformation});
ExportDataTask taskED= new
   ExportDataTask(dbManager.getConnection(), driver, db, schemaLoc);
taskED.go();
while (!taskED.isDone()){
   Thread.sleep(100);
}
OntModel data=taskED.getDataOntology();
```

Listing 5.2: Data export

Schema Import: Besides its data and schema export functionality, the Relational.OWL application is also capable to import the Relational.OWL representation of a database into another database, i.e. to transform a Semantic Web representation of a database back into its original, relational format. The code snipped required to transform a schema ontology back into a relational schema can be reviewed in Listing 5.3.

```
String importSchemaPath = pathToSchema;
DatabaseManagerFactory.getDbManagerInstance(connInformation);
ImportSchemaTask taskIS = new
  ImportSchemaTask(dbManager.getConnection(), driver, impSchemPth);
taskIS.go();
while (!taskIS.isDone()){
   Thread.sleep(100);
}
```

Listing 5.3: Schema import

Data Import: Importing the Semantic Web representation of data items into a relational database requires a corresponding database schema to be available. Please note, that it is irrelevant how this schema was created, although it will most likely be created using the technique presented above. The data import task itself is then performed quite analogously to the schema import presented above, i.e. a database connection is established and subsequently a thread performing the data transformation and import is started (cf. Listing 5.4).

```
DbManager dbManager =
   DatabaseManagerFactory.getDbManagerInstance(connInformation});
ImportDataTask taskID = new
  ImportDataTask(dbManager.getConnection(), drvr, schemPth, datPth);
taskID.go();
while (!taskID.isDone()){
   Thread.sleep(100);
}
```

Listing 5.4: Data import

5.1.2 Relational.OWL with XSLT and XQuery

After having introduced the Java-based Relational.OWL application in Section 5.1.1, we now present an alternative implementation of the Relational.OWL technique. This version of the Relational.OWL application dos not require external applications, but works completely within the relational database system. Since it is implemented using XML technologies like XSLT [XSL99] or XQuery [BCF+05], the only requirement is that the relational database system supports this kind of XML functionality.

The corresponding functions are published under the GNU GPL and can be downloaded from `http://sourceforge.net/projects/relational-owl/`. A more detailed documentation is available at [Mat06].

5.1.2.1 Introduction

The most important advantage of implementing Relational.OWL within a relational database is, that once installed, anybody having access to that database may benefit from the extended functionality, i.e. nobody needs to install or address an external application anymore. The disadvantage of using stored procedures to create the Relational.OWL representation of a database is the portability. Since database systems from different vendors use different programming languages for their stored procedures (e.g. PL/SQL or Transact-SQL), the portability of the required stored procedures could become a challenging task.

Although most relational database management systems already support Java-based stored procedures and we could have tried to call the Java-based application (cf. Section 5.1.1) using a stored procedure, this would have been a tremendous overhead, since whenever a stored procedure would have been called, the underlying Relational.OWL application would have established again a connection to the local database, in order to perform the data or schema extraction task. Having the possibility to use XML techniques like XSLT or XQuery in the Oracle 10g, Release 2 database management system, we implemented this functionality using these techniques. The advantages of this approach are obvi-

ous, since using standard XML techniques like XSLT or XQuery, we are easily able to transform the system-specific XML representation of the database into an arbitrary XML document, e.g. its Relational.OWL representation using the RDF/XML syntax. Since most database vendors are willing to support this advanced XML functionality, an easy portability to those systems will be most likely.

5.1.2.2 Implementation

Since the Oracle 10g database system supports the XSL Transformations language and XQuery, we have implemented the Relational.OWL functionality using both techniques and were hence able to compare both implementations with respect to their performance, revealing that the XSLT version of the stored procedures are considerably faster than the XQuery implementation. The detailed performance results can be reviewed at [Mat06]. We now introduce shortly the stored procedures required for creating the Relational.OWL representation of the database.

Since both, XSLT and XQuery required XML documents as a starting point, we first need a raw XML view of the schema and data components to be transformed. This functionality is implemented in the `DATA_AS_RAW_XML()` and `METADATA_AS_RAW_XML()` functions, which rely on the XML transformation functionality provided by the underlying database system. They are both invoked by the corresponding XSLT and XQuery procedures.

XQuery: The XML query language XQuery [BCF+05], which is currently a candidate recommendation at the W3C (cf. [Jac05]) and will be a de-facto standard for querying XML documents provides the functionality to create arbitrary XML documents from the results of a query. This can be used to transform the raw XML document as provided by the XML functions mentioned above to a valid RDF/XML document corresponding to the Relational.OWL representation of that specific database. The according function, which performs the schema conversion into the schema ontology of the database is `METADATA_AS_OWL()`, whereas the function which creates the instances of this ontology, i.e. converts the data items is called `DATA_AS_OWL()`.

XSLT: The XSL Transformations language XSLT [XSL99], which was already recommended by the W3C in 1999, is an XML stylesheet language which transforms a given XML document using matching rules into a target XML document. Hence, we are again able to transform the raw XML representation resulting from the `DATA_AS_RAW_XML()` and `METADATA_AS_RAW_XML()` functions to RDF/XML files, corresponding to the Relational.OWL representation of the database. The functionality to create the schema and data

representations is implemented in the `METADATA_AS_OWL_WITH_XSLT()` and `DATA_AS_OWL_WITH_XSLT()` functions correspondingly.

Each of the four functions mentioned above, which transform the raw XML documents into RDF/XML documents, returns an `XMLType` value (cf. Section 2.1.2). If it is required to export this data into a file, the functions `EXTRACT_METADATA_INTO_FILE()` and `EXTRACT_DATA_INTO_FILE()` can be used. A detailed description of the functions, including their parameters is provided in [Mat06] and at the URL specified above.

5.2 RDQuery

One of the main drawbacks of the Semantic Web is the lack of semantically rich data, since most of the information is still stored in relational databases. We now present RDQuery, a wrapper system which enables Semantic Web applications to access and query data actually stored in relational databases using their own built-in functionality. RDQuery automatically translates SPARQL and RDQL queries into SQL. The translation process is based on the Relational.OWL representation of relational databases and does not depend on the local schema or the underlying database management system. A similar introduction to RDQuery has also been published in [PZC06].

5.2.1 Introduction

With his vision of a Semantic Web, Tim Berners-Lee inspired the database and knowledge representation communities to build up the next generation Web. Despite its sophisticated technologies like RDF [MM04] and OWL [Mv04], the Semantic Web still has to face its major drawback, the lack of data. In fact, data is usually still stored in relational databases where it cannot be accessed directly by Semantic Web applications. If these applications need to query relational data, they usually have to do this using SQL and create a corresponding mapping between the relational and the semantic models on their own. Due to the fact that such mappings have to be created manually, the consequences are obvious: different applications could map identical data extracted from the same database to different concepts of the Semantic Web. Consequently, a well-defined mapping of relational to semantic data is required.

Although we can convert the schema of a relational database automatically into an RDF/OWL ontology and represent the corresponding data items as instances of this data source specific ontology (cf. Section 3.1), barely a database is static. Consequently, this data and schema extract may rapidly become outdated. Indeed, a schema or data extraction could be initiated, whenever a data or schema modification occurs within the database. Nevertheless, dealing with dynamic data sources, a direct access to such data sources would be preferable.

In this section we introduce RDQuery, an application which automatically translates Relational.OWL specific RDQL and SPARQL queries into their SQL correspondents. RDQuery is hence able to provide Semantic Web applications real-time access to relational databases using their own RDF query languages (RDF-QL).

5.2.2 Query Translation

RDQuery is a wrapper system which makes relational databases accessible for Semantic Web applications using an RDF-QL. RDQuery currently supports RDQL [Sea04] and its successor SPARQL [PS06b]. Nevertheless, RDQuery may easily be adapted to future developments adding specific parsers for other query languages. Figure 5.2 gives an overview of the RDQuery system architecture and depicts the path passed by a query until it reaches the relational database as its destination.

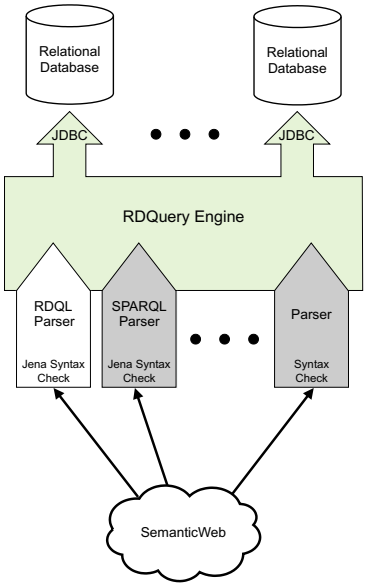

Figure 5.2: RDQuery system architecture

First, the syntax of the query is validated and its relevant parts (e.g. the `WHERE` clause) are extracted using the built-in syntax checker of the JENA Framework [Jen06]. Thereupon, the relevant parts of the RDQL or SPARQL query are once again parsed using an own JavaCC-based (Java Compiler Compiler) [Jav06] grammar, in order to detect the properties of the query. Based on this information, the corresponding SQL query is built up. The resulting query is then executed and processed on the original database without having to translate the

original database into a Relational.OWL representation, which thus only exists virtually.

The query translation is based on the results presented in Section 3.2.1 and Section 3.2.2, where we examined possible RDQL and SPARQL correspondents for the basic expressions of the relational algebra. Each of the five main operations $\{\sigma, \pi, \cup, -, \times\}$ of the relational algebra has characteristic appearances within a Semantic Web query, making it possible to identify them and build up a corresponding SQL query, as long as the queries match the Relational.OWL representation of a database.

For instance, the triple `{?x dbinst:TABLE.COLUMN 'value'}` expresses that the objects represented by the free variable `?x` shall be restricted to the objects with their attribute `TABLE.COLUMN` matching `value`. This constraint obviously corresponds to a selection, i.e. the `WHERE` part of an SQL query. On the other hand, the `FROM` clause of an SQL query can be composed of the `{?x rdf:type tablename}` triples contained in the query, since they specify the object class, i.e. the tables, the objects originally came from. The remaining operations of the relational algebra can be mapped correspondingly.

Example: The SPARQL query

```
CONSTRUCT {?a ?b ?c}
WHERE {{?a ?b ?c}.
       {?a rdf:type dbinst:customers}.
       {?a dbinst:customers.City 'Berlin'}.
   FILTER (?b=dbinst:customers.ContactName)}
```

is automatically recognized by RDQuery as the SPARQL correspondent of a selection, followed by a projection. It thus translates the given query automatically into the following SQL query:

```
SELECT customers.ContactName
FROM   customers
WHERE  customers.City = "Berlin"
```

The user may opt to receive an RDF processable representation of the query result, after it was executed at the corresponding relational database. This feature of RDQuery is of particular importance for Semantic Web applications using a non-closed query language like RDQL, since they are instantly able to process the query result without having to convert the possible variable bindings, they would usually receive, into valid RDF.

The whole query transformation process is identical for any relational database and does not depend on the local schema or the underlying database

management system. Nevertheless, the queries have to match the instances of the Relational.OWL ontology. For a detailed description on how to simulate the main operators of the relational algebra in RDQL and SPARQL, we again refer to Sections 3.2.1 and 3.2.2.

5.2.3 Usage

RDQuery gives the user two possibilities to translate Semantic Web queries into their SQL correspondents. Whilst the experienced user is able to incorporate the framework into his Java-based (Semantic Web) application, RDQuery may also be tested using its Java-based user interface.

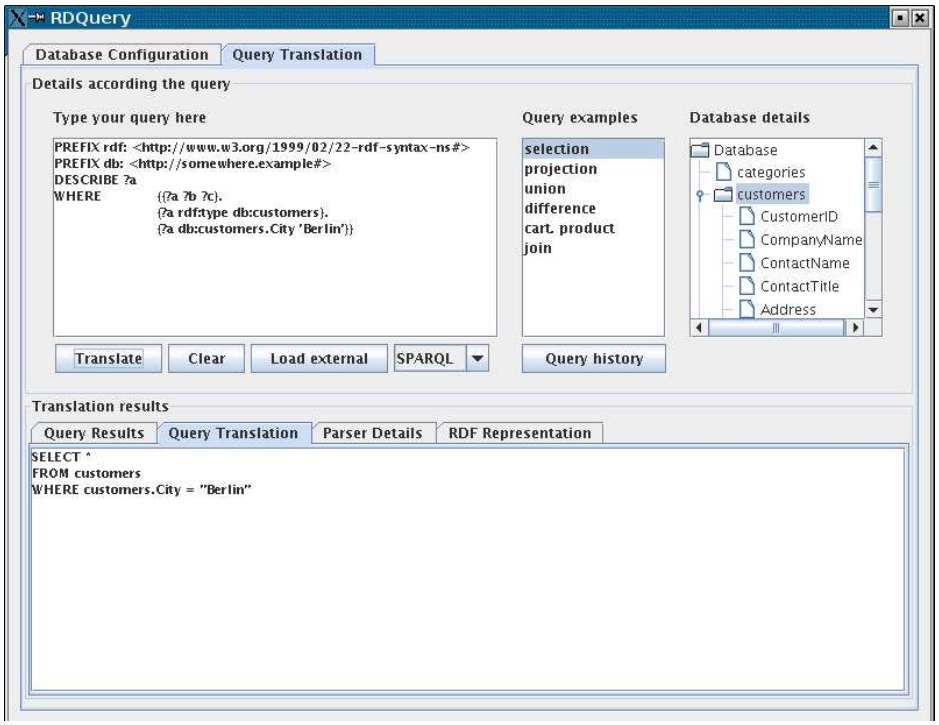

Figure 5.3: Sample query translation using the GUI

The GUI (cf. Figure 5.3) enables interested users to interactively query relational databases using one of the RDF-QL currently implemented in the system, i.e. RDQL or SPARQL. The users may not only review the resulting SQL query, but follow the complete translation process, including the parser details, resulting SQL query, the SQL query result, and its RDF correspondent. Furthermore, the users can access their own query history and get a general idea of the tables stored in the corresponding database.

Does a user or Semantic Web architect opt to include the functionality of RDQuery into his own application, he may use the API of RDQuery. The user

is hence instantly able to query any relational database using current Semantic Web technologies, pretending the relational database to be actually a part of the Semantic Web. The application may hence perform any data processing or reasoning tasks, without actually noticing, that the data is stored in and modeled for a relational database.

The following code example shows, how to translate an RDQL query into SQL, without having to query the underlying database. Before initiating the actual translation, the `SimpleTranslation` class has to be instantiated with the corresponding RDQL query as an argument.

```
SimpleTranslation myTranslation =
           new SimpleTranslation("SELECT ?x, ?y, ?z ...");
myTranslation.translate();
```

After the query has been successfully translated into SQL, the corresponding statement may then be accessed with

```
String sqlQuery = myTranslation.getSQLQuery();
```

A more detailed documentation on how to use RDQuery is available in [Zlo05] or can be downloaded from `http://www.sourceforge.net/projects/rdquery/` together with a current version of the framework. RDQuery is published under the GNU GPL.

5.3 DÍGAME

Modern intra- and inter-enterprise collaboration requires access to information spread over multiple autonomous and heterogeneous data sources. In this section we present how a semantic data and schema representation like the Relational.OWL representation of a relational database can enhance the communication of a loosely coupled multidatabase architecture. The architecture presented in this section achieves a reasonable tradeoff between autonomy and information sharing among both, permanently available and volatile data sources. Each data node decides autonomously which kind of information to share. Data availability, query performance, and up-to-dateness on each participating data node is improved using a push-based replication strategy, which propagates data modifications over multiple nodes. Most of the work presented in this section has also been published in [PP04, PPC04a, PPC04b, PPC05].

5.3.1 Motivation

Since the first centralized databases found their way into the enterprises in the late 60s, the needs and requirements have changed towards a more distributed

management of data. Today there are many corporations which possess a large amount of databases, often spread over different regions or countries and generally connected to a network. These local databases typically raised in an autonomous and independent manner fitting the special needs of the users at the local site. This leads to logical and physical differences in the databases concerning data formats, concurrency control, the data manipulation language, or the data model [LA86]. It is crucial for a company to keep track of its distributed data in such a heterogeneous environment. Cooperating departments need shared access to this data, to be able to increase their productivity. Multidatabases were introduced for this reason, in order to integrate data from heterogeneous sources [SL90].

One of the main challenges in the integration of data in such environments is the autonomy of the participating data nodes. This autonomy implies the ability to choose its own database design and operational behavior. Local autonomy is tightly attached to the data ownership, i.e. who is responsible for the correctness, availability, and consistency of the shared data. Centralizing data means to limit local autonomy and revoke the responsibility from the local administrator, which is not reasonable in many cases. A federated architecture for decentralizing data has to balance both, the highest possible local autonomy and a reasonable degree of information sharing [HM85, Con97]. Hence, the architecture of a company wide information system has to be applicable to the data policy of the company and vice versa. To be more precise, the question of data ownership determines the composition of the company wide information platform, while it has to ensure a high level of consistency and fail-safety.

In this section we describe the DÍGAME architecture, a **D**ynamic **I**nformation **G**rid in an **A**ctive **M**ultidatabase **E**nvironment, which connects heterogeneous and autonomous data sources to support loosely coupled intra- and inter-enterprise collaboration. We have enhanced the multidatabase architecture of Heimbigner and McLeod [HM85] with Peer-to-Peer (P2P) concepts to offer a flexible information grid with high data availability to provide each participating node of this grid with all the data required. Extending the approach of Heimbigner and McLeod, our architecture enables the sharing of information among both, permanently available and volatile data sources (e.g. mobile databases [Bar99]) without any central component. For that we include a push-based replication mechanism which propagates data modifications over multiple nodes using a semantically rich data and schema representation format. Thus, we are able to ensure flexible interconnectivity and high availability, even if the original data source is temporarily unreachable. Additionally, the replication of data increases query performance, since we do not have to query remote data sources. An information sharing environment, which comprises the information shared by interconnecting heterogeneous and autonomous data peers using our architecture, shall in the following be referred to as an information or data grid [CFK+00].

The remainder of this section is organized as follows. In Section 5.3.2 we start describing the architecture of our dynamic information grid with an overview of

the basic functionality. Afterwards we discuss the major characteristics in Section 5.3.3, followed by a description of our first wrapper prototype in Section 5.3.4. Section 5.3.5 discusses related work and Section 5.3.6 concludes and draws up future work.

5.3.2 DÍGAME Architecture

5.3.2.1 Basic Functionality

We now draw up the basic functionality of our enhanced multidatabase architecture using a motivating example.

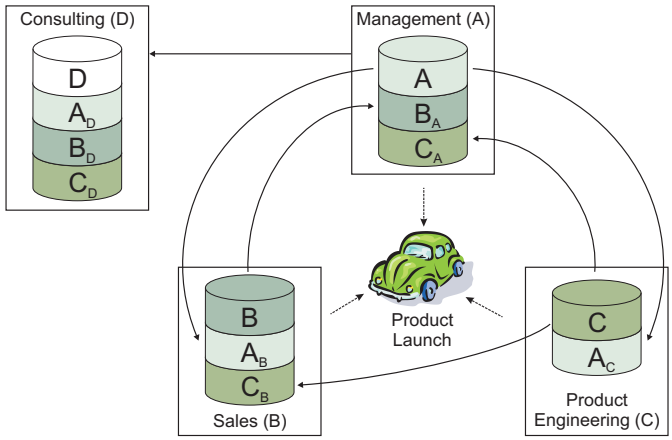

Figure 5.4: Collaborative work realized with DÍGAME

Consider a worldwide operating company planning the launch of a new product (Figure 5.4). We assume that there are three departments involved in this business process: the executive board (management), the sales office, and the product engineering department. Each department manages its own database to store the information for which it is responsible in an autonomous way. The management produces basic data of the product (A) including deadlines, descriptions, workflows, and additional objectives. This management information is substantial for the further product development and the work in the participating departments. The product engineering department uses a predefined part of that management data (A_C) as basic conditions for the concrete implementation and technical realization of the product. Local applications create additional data which has to be stored separately (C). According to the product engineering the sales department enriches the authoritative management data (A_B) with concrete concepts for the upcoming product launch (B). Furthermore concrete development plans of the product engineering are required to prepare sales strategies (C_B). Both, sales and product engineering departments, concretize the strategic guidelines of the management in their specific assignment. To keep track of the

costs and the progress of the project, it is indispensable for the management to access the product engineering and sales department's relevant information just mentioned (B_A, C_A).

Basically there are two different techniques for providing the participating departments with the required data. Contrary to the commonly used method querying the data sources actively, our approach uses push-based replication initiated by the data source. Referring to our example, the executive board gets data updates whenever changes occur in the sales and/or product engineering databases rather than having to request for updated data items continuously. If the subscribing department requires individual delivery strategies, we are furthermore able to provide data updates periodically, i.e. in an aggregated way.

Sharing data within this company using our architecture is realized using a push-based replication strategy to improve data availability, query performance, and up-to-dateness on each participating data node. Hence, the data source actively propagates data updates to relevant peers, which are herewith able to maintain an up-to-date replica of the imported data.

For example, the creation of a new replica of management data on department B is realized as follows: each data source of the departments A, B and C is wrapped by a source-specific wrapper component. These wrappers build up a communication layer, which enables the departments to interact pair-wise using a common protocol. This union adopts Peer-to-Peer concepts and operates without any central administrative instance. Due to these characteristics the combination of such a data source and its related wrapper component can be named as a (data) peer.

The administrator of peer A makes a subset of its own local data accessible using the administrative interface of the wrapper component. The export schema [HM85] created this way is managed by the wrapper component and specifies the information that the department is willing to share. The information concerning the access control to local data by remote peers is attached to the export schema. Peer B is now able to import the data into its local database subscribing to a specific part (A_B) of the published data, i.e. the data required by the department. During this subscription process the data target (subscriber) informs the data source (publisher), which subset of the export schema it is willing to import. The data stock A_B is then transferred to the subscriber to perform an initial filling.

If a data or schema modification is detected by the wrapper of the publisher, all relevant subscribers have to be informed. To determine whether the subscribers, including peer B, have to be notified about this modification, the wrapper queries all export schemas in the repository. The modified data or schema items are then pushed actively to the relevant subscribers using a semantically rich representation format. Each data peer is herewith able to maintain an up-to-date replica of the data and schema items required by local applications.

Now the management department has decided to involve an external consult-

ing group D to analyze and optimize the productivity within corporate workflows. Therefore the consultants need access to the entire management data, including the data of departments B and C. Instead of negotiating separate data exchanges with every single department, our architecture enables the consulting group to obtain all data required from only one data source, the management department. This can be realized, since our architecture supports the sharing of data imported from other nodes. Please note, that the export of imported data must explicitly be allowed by the administrator of the management department. After the consulting has subscribed to the entire management data, data updates in B and C are propagated to A as usual. Node A delivers updates on its own data stock and additionally those coming from nodes B and C to its subscriber D. Due to this characteristic, node A becomes a Data Hub for the consulting group according to the Link Pattern Catalog introduced in Chapter 6.

We distinguish two different types of update propagation: *direct* and *indirect updates*. After an update is detected on local data of a data source, it is propagated to the relevant subscribers. Referring to our example, B gets direct updates, whenever modifications occur on the data stock of node C. If a node explicitly shares a previously imported data stock, its modifications are in turn propagated to other subscribers. Referring to our example, node A shares previously imported data from peers B and C, which is subscribed by the consulting group node D. If an update occurs on B or C, it is first propagated to node A, which in turn propagates it to its subscriber D. This sequence of update propagation is called a *cascading update*.

Further partners may join this collaboration at any time. In fact, each peer can be provided with any data concerning the product launch stored in one of the collaborating data nodes without interfering with existing data flows. The data source maintained by the partner can on the other hand be easily connected to the existing data grid sharing its own data. If a peer is no longer willing to share its data, it can easily be removed from the data grid, notifying all its subscribers to remove the replicas from their local data stocks. The support of this temporary collaboration makes our DÍGAME architecture particularly suitable for virtual cooperations.

5.3.2.2 Components of the Architecture

In this section we discuss the components of our DÍGAME architecture (Figure 5.5). The data grid $DG := (P, C)$ created by our architecture is a directed graph, which consists of a set of peers $P := \{p_1, ...p_n\}$ and a set of connections C, where a connection $c = (p_i, p_j) \in C$ links exactly two peers, representing a data flow from p_i to p_j.

As already mentioned, each peer consists of a component database and a corresponding wrapper component. These components which are both required for

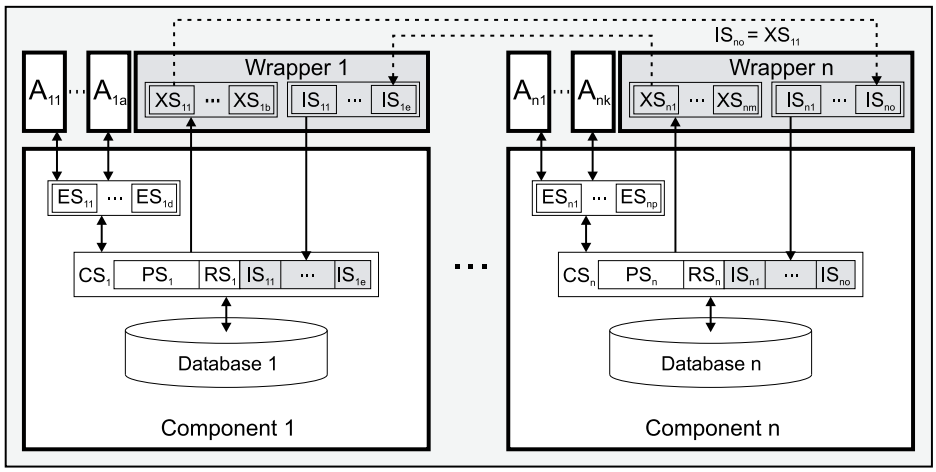

Figure 5.5: DÍGAME architecture

establishing data flows between communicating peers are described in the following:

Wrapper: The core of our data grid is the wrapper component, which provides a uniform interface to the heterogeneous component databases. It is responsible for negotiating and establishing communication among peers and coordinates the data and schema exchanges after a communication channel has been set up. Each wrapper maintains a repository in its corresponding data source to store information about subscribers, export and import schemas, access control lists, and delivery schedules.

Each wrapper has to realize two major tasks: exporting and importing data and schema items. To export local data from a peer p, a set of export schemas $\mathcal{XS}_p := \{XS_{p1}, ..., XS_{pi}\}$ is maintained by the wrapper of p. To allow indirect updates, those export schemas have to be based on the entire conceptual schema CS_p of the database, excluding RS_p, the schema of the repository stored in p, i.e. $\forall XS \in \mathcal{XS}_p : XS \subseteq CS_p \setminus RS_p$. They are required to determine, which peers have to be informed about data modifications. Since exporting peers actively propagate the data and schema to relevant subscribers, they must be able to detect modifications on their local data stock. Earlier research proposes several mechanisms helping a wrapper to monitor data modifications [TC97]. If there are triggers of underlaying database systems available, they should be used, particularly their enhanced functionality given by recent developments in database systems [PC05c].

To import data from a remote peer p, the wrapper on a peer q ($p \neq q$) maintains a set of import schemas $\mathcal{IS}_q := \{IS_{q1}, ..., IS_{qj}\}$, where

$$\forall q \in P \; \forall IS \in \mathcal{IS}_q \; \exists_1 p \in P \; \exists_1 XS \in \mathcal{XS}_p : IS = XS \wedge p \neq q \tag{5.1}$$

and

$$\forall p \in P \; \forall XS \in \mathcal{XS}_p \; \exists_1 q \in P \; \exists_1 IS \in \mathcal{IS}_q : XS = IS \wedge q \neq p \; . \tag{5.2}$$

After the initial import of subscribed data, each data and schema modification propagated by remote peers is reproduced locally in the workspace of the wrapper.

Autonomous Component Databases: According to the Three Schemas Architecture and the architecture for loosely coupled multidatabases [HM85], each component database on a peer q contains several types of schemas (see Figure 5.5). The private schema PS_q stores data, which is locally produced and maintained. It is controlled exclusively by the local database administrator. Other peers do not have direct access to this data. Besides the private schema, the conceptual schema CS_q comprises the disjoint union of the import schemas and the repository mentioned above, i.e. $CS_q := (\dot{\bigcup}_{IS \in \mathcal{IS}_q} IS) \cup PS_q \cup RS_q$, where $IS \cap PS_q = \emptyset$. Local applications $A_{q1}, ..., A_{qf}$ can now access and process the data of the conceptual schema excluding the repository information as usual using a set of external schemas $\mathcal{ES}_q := \{ES_{q1}, ..., ES_{qd}\}$. The only limitation is the read-only access to data derived from the imported schema.

Please keep in mind that the imported data and the repository are exclusively managed by the wrapper component and should never be modified by the local administrator or applications, although this would be possible due to the local autonomy. In fact, future implementations could support such multi-master replication techniques.

5.3.3 Characteristics

In this section we discuss the major characteristics of our DÍGAME architecture including the advantages and limitations related to its implementation.

Autonomy and Heterogeneity: Our architecture is based on the concept of loosely coupled multidatabases of Heimbigner and McLeod [HM85] using import and export schemas for data exchanges. The aim of this architecture is to achieve a feasible trade-off between local autonomy and a reasonable degree of information sharing. A data source is basically free to decide on its own level and form of participation. This includes the ability to decide which data it is willing to export, which data is imported, and during which periods services are provided.

The wrapper component interacts with the data source via standardized interfaces or query languages. Thus, it acts like a local application from the point of view of the database system preserving especially its execution and communication autonomy [SL90].

Our architecture supports the integration of principally any kind of data source using a wrapper component tailored to that specific data source. The wrapper provides an uniform interface for the DÍGAME system, where communication is performed using a standardized protocol and exchange format.

No Central Authority: Any information sharing environment based on our DÍGAME architecture interconnects autonomous and previously isolated data peers. Each participating data node keeps full control over its own data, i.e. there is no central authority imposing certain restrictions. Contrary to other approaches like [YPK03] we do not use any central component, where publications or subscriptions are managed. In our system, peers subscribe directly to data published by other nodes. The information on the data offered is not managed centrally, but stored exclusively on the corresponding peers.

Wrapper organized similar to P2P systems: We have enhanced the multidatabase architecture with P2P concepts. The wrappers in our architecture interact similar to classical P2P networks. Data exports and imports are exclusively negotiated pair-wise, whereas each peer is basically able to interact with any number of data nodes. The entire communication is realized without any central authority, resulting in a network of self-responsible peers, where members are basically able to join or leave at any time.

Replication: The replication of data is one of the main features of our DÍGAME architecture. Data availability is improved in the information grid allowing a data stock to be directly or indirectly replicated over multiple peers. This means, that required data is accessible, even if the original data node is temporary unavailable. Furthermore query performance is increased, since all the required data is stored locally.

The refreshment strategies for updating the replicas depend on the application field. We are basically not limited to a single delivery schedule, but able to provide specific replication strategies depending on the needs of each subscribing peer. Generally, the preferred delivery schedule is an immediate propagation of updates, but other possible delivery schedules can be, but are not limited to periodical or even aggregated propagation. The replication is managed by the wrapper component, which holds information about each subscribing database and its corresponding delivery schedule in its corresponding repository.

Due to the replication supported by our architecture, each single data peer provided with data updates maintains a replica of remote data locally, which balances both, up-to-dateness and a reasonable effort. The replication strategies provided by DÍGAME use *lazy replication* protocols with one single master and multiple read-only replicas [OV99]. Each update transaction is first committed at the master and afterwards propagated to each relevant slave.

Push-based Protocol: A further central characteristic of our architecture is the push-based propagation of data and schema modifications to subscribing peers. At first a data peer subscribes to data offered by a data source, whereupon it receives once a complete copy of the requested data. Afterwards the data source pushes all relevant updates directly to the subscribers according to their specific delivery schedule. This modifications are passed on to further subscribers using indirect updates, until all replicas are updated. Each peer maintaining a replica

of remote data is herewith able to access data, which is as up-to-date as possible, even if the original data source is temporary not available.

If a replica can not be updated, because a subscriber is currently not reachable, we have decided to include a pull-based fallback mechanism into our architecture. After the communication has been reestablished, the data target can then actively query the data source whether data updates have occurred since their last contact. Thereupon lost updates are propagated once again to the data target.

Standardized Exchange Format: The dynamic interconnectivity of data peers requires a standardized exchange format, suitable for both, data and schema representation. Using knowledge representation techniques we can guarantee that every single data peer understands data and schema updates without explicitly arranging an exchange format. The additional integration of identifiers for data items (cf. Chapter 4) within the data exchange process simplifies data maintenance, especially if data is imported from multiple sources. This meta information may furthermore be useful for detecting and solving conflicts within the data.

We have decided to use a Semantic Web-based data and schema representation format, since it provides several advantages over classical (semi)structured exchange formats like XML. Due to the short availability of potentially any peer, the negotiation of an exchange protocol for data and schema items becomes quite challenging. An exchange format, which can be understood instantly by all exchange partners would be more useful. Hence, using a Relational.OWL-based representation format, remote databases are instantly able to understand each other without having to arrange an explicit exchange format — the usage of a common ontology is enough.

Based on a meta representation of (relational) databases we can describe the schema of virtually any database. Thereupon the schema representation itself can be used as a novel schema ontology, to base the representation of the actual data on. The result is a three layered model with the Relational.OWL ontology on the bottom. The layer above stands for the concrete schema ontology, which itself is based on the Relational.OWL ontology. The representation of the data, which in turn is based on the schema ontology is placed on the top-most layer. This flexible and powerful technique is only possible due to the possibilities given by *OWL Full* to interpret an instance of a metamodel as a novel ontology. A detailed discussion on this topic can be found in Section 3.1.

Local Integration: As already mentioned above, each peer may subscribe to multiple data sources. For each subscription it obtains an exact copy of the relevant remote data and schema items. Since we do not have a global schema, the imported data is integrated individually following local integration strategies, which are not provided by our DÍGAME architecture. Having all required data stored in the local database, we are particularly able to associate local and *remote* data with integrity constraints provided by the database, e.g. foreign keys.

Furthermore index structures can be created on imported data to optimize data access according to local query requirements.

5.3.4 DÍGAME System Design

Now we give a brief overview of the design of our first DÍGAME wrapper prototype. The components required are depicted in Figure 5.6. Our system is divided into two conceptual layers: a source specific and a source independent layer. Both layers interact exclusively using Java objects. The source specific layer contains all classes and packages, which are customized to the specific data source. This layer has to be implemented for each type of data source and provides a uniform interface for the classes of the source independent layer. Thus, new data sources can be supported exchanging solely the source specific components, leaving the remaining components unchanged.

The main component of our system is the DÍGAME Manager. It initializes the remaining components and coordinates the entire system flow. At first, it is responsible for the creation and management of import and export schemas, which are set up by the administrator using a corresponding user interface and stored in the repository of the wrapper. Thereupon, the DÍGAME Manager automatically handles subscription and unsubscription requests to shared data from remote peers and coordinates the data exchange during data import and export processes. After a data modification has been reported, it determines the peers which have to be provided with that updated data and initiates the corresponding data transfers. Outgoing data is compressed by the DÍGAME Manager using the Compression Unit to reduce network traffic.

The entire communication with the data source is handled by the Data Handler. It consists of a set of functions providing a uniform interface to the data stored in that specific source. This interface is used by the DÍGAME Manager and the event detection packages to access the data and the repository, which are stored both directly in the data source. Thus, the access to both, the storage of data and metadata are entirely managed by the Data Handler component. As a result, all source specific properties, i.e. data model, query language, or operating system are hidden from the components of the source independent layer.

The Data Handler supports three types of interactions: *data requests*, *repository requests*, and *event notifications*. The DÍGAME Manager component submits *data requests*, whenever a new peer subscribes to the data or data updates are received from remote peers. In the first case, data items have to be transformed into the Relational.OWL-based exchange format, in the latter from RDF/OWL into the source specific data format. These conversions are realized by the Relational.OWL converter, closely attached to the Data Handler. Unlike the data, information stored in the repository is only used internally and thus, responses to *repository requests* are not transformed into their Semantic Web representation.

Changes on the local data stock are signalized to the Data Handler via *event*

notifications by the event processor. These events are either detected by the notification interface or the event monitor. If a data source supports extended triggers, the notification interface is directly invoked by database triggers to notify the event processor about local data modifications [PC05c]. Otherwise, such events have to be detected by the event monitor, which therefore periodically scans the local data stock.

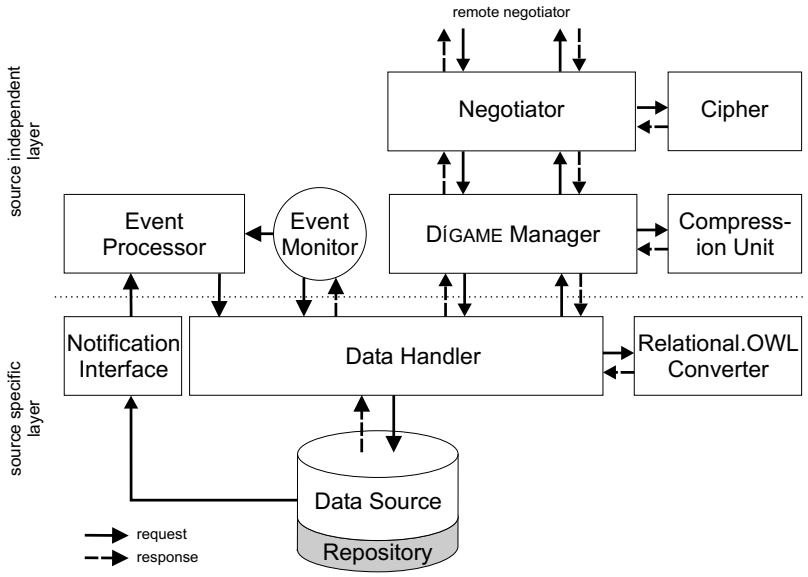

Figure 5.6: Design of the DÍGAME wrapper

A communication between two peers is established using the Negotiator. It interacts with remote peers using a special protocol to establish a secure communication channel. After the successful authentication and authorization it is used by the DÍGAME Manager for interacting with remote peers. The communication is encrypted using the Cipher component of our system.

5.3.5 Related Work

The first generation of grid computing emerged in the mid 1990s with the demand for high performance applications, which could not be satisfied by single computers. De Roure et al. [DBJS03] divide the evolution of grid computing into three generations: the first generation with its primitive architecture, which tried to distribute computing onto different computers a trivial way. With the second generation of grid computing middleware systems emerged, and finally the current third generation tries to facilitate global collaboration.

Simultaneously some efforts arose to use distributed resources for information retrieval. Although the *Information Grid* of Rao et al. [RCJ+92] is focused on giving an integrative user interface for distributed information, this approach can be

seen as an early forerunner of the so called *Data Grid* [CFK+00], a specialization and extension of grid computing. Its intention is to create an architecture of integrated heterogeneous technologies in a coordinated fashion. Although Chervenak et al. act on the assumption of a heterogeneous conglomerate of data sources, they force the introduction of a centralized metadata repository, e.g. an LDAP directory [Sto01]. This aim is quite catchy especially in a grid consisting of completely autonomous databases changing their schemas frequently. Although we admit that a global metadata repository would simplify many of the challenges, we abstain from that effort of re-centralization, as it causes many difficulties, e.g. every schema change has to be replicated to the global schema directory. The effect is a single point of failure, exactly the opposite of what we wanted to construct. We thus prefer to keep the databases as they are: autonomous, loosely coupled, and without a single point of failure.

With the raise of filesharing systems like Napster or Gnutella [CG01] the database community started to seriously adopt the idea of P2P Systems to the formerly known loosely coupled database systems. Contrary to the data grid, P2P database systems do not have a global control in form of a global registry, global services, or a global resource management, but multiple databases with overlapping and inconsistent data. These P2P databases resemble heterogeneous and distributed databases, also known as multidatabases [HJKS06, BGK+02, GHI+01]. Currently the database community makes a great effort in investigating P2P databases. Worth mentioning is especially the *Piazza* [HIMT03] project, where a P2P system is built up with the techniques of the *Semantic Web* [BLHL01] with local point-to-point data translations rather than mapping to common mediated schemas or ontologies. Halevy et al. focus on processing and rewriting queries on XML data throughout multiple peers. Contrary to this approach, we deal mainly with relational data and do not have a global schema, since every peer may have its own import-/export-schema combination. As a result every peer has its own integrated schema as basis for queries. Beyond this, Piazza can only deal with data updates as long as the peers are online. As soon as one peer is disconnected from the network data consistency cannot be guaranteed any more. Like in most P2P approaches, peers may not have all the information required for their queries stored locally, so they have to deal with query and result rewriting. This is superfluous in our architecture, since all the data required is cached on that peer. Similar to our approach, Piazza allows only data updates in its origin. Hoschek follows a quite different approach [Hos02], since his goal is to let the loosely coupled databases appear to be a single data source and thus has to deal with distributed query processing. For a more general glimpse on data mappings in P2P systems see [KAM03].

Our strategy allows data to be exchanged among distributed databases connected through a lazy network. This means, that although a running network may not be guaranteed and thus some data broadcasts may be lost, the system heals itself. This challenge resembles the problems known from environments

with mobile databases. Current research covers synchronous mobile client synchronization, i.e. data changes are propagated periodically (every t seconds) and not just in time of the data change. Current systems have two main problems which arise with the synchronous replication: clients have to be contacted every t seconds, no matter if changes have occurred and in the worst-case changes have to be delayed for t seconds. For a more detailed discussion we refer to [Har02], as most *push-based* technologies base on the idea of *broadcast disks* [AFZ96, AFZ97]. In contrast to the broadcast disks, our model ensures that data is only broadcasted to the clients when changes occur, unless the communication between both peers crashes. Hence our approach resembles a *push-based* system with a *pull-based* fallback, similar to [AFZ97] with the major difference that our approach is not based on broadcast disks, but on the Observer's Pattern (see below).

There has been much effort in the research of better and more efficient techniques for data propagation, caching, and replication. The evolution of these methods started with early papers like [JT75] for classical database systems and goes to more recent publications for mobile clients like [Bar99, BI94, Har02]. For a classification of database replication techniques see [WPS+00]. As mentioned above, we have decided to use a push-based replication strategy, which resembles the software engineering's *Observer-Pattern* [GHJV95]. This pattern gives us a prototype of how to notify all interested databases about data updates [Har02]. This communication is only started, if a data update has occurred and a database is interested. In consequence, data broadcasts are minimized.

Following the argumentation in [GHOS96] and [CFK+00] our model provides only single-master replication, the only guarantor for data stability and clear defined data flows.

Most of the research on active multidatabases has been done concerning global integrity. Chawathe et al. [CGMW96] propose a toolkit for constraint management in loosely coupled systems. Additionally the idea of Gupta and Widom to optimize the testing of global constraints by local verification is worth mentioning [GW93]. Conrad and Türker [TC97] sketch a more general architecture for an active federated database system. They extend a multidatabase system by ECA-Rules to preserve consistency. A main challenge hereby is to detect local events, especially schema and data modifications, which is commonly done by a software module for each data source, i.e. a monitor or wrapper component. Basically two approaches are therefore proposed: Conrad and Türker use the event detection ability of the underlying subsystem, while Blanco et al. [BIPG92] use the operating system to signal schema modifications by directly observing changes to the data(base) files.

5.3.6 Discussion and Future Work

In this section we presented the DÍGAME architecture, which connects heterogeneous and autonomous data sources creating a dynamic information grid. This

architecture enhances the well-known multidatabase architecture with P2P concepts, in order to support dynamic intra- and inter-enterprise collaboration. Local administrators decide themselves on their level of participation, since the local autonomy is preserved.

Data provided by other peers can be subscribed and integrated into the local database as needed. The data source actively propagates changes on the subscribed data and schema items to the relevant peers via a Semantic Web-based representation format resulting in a replication of the data and schema items demanded locally. Peers participating in the data grid interact pairwise and are instantly able to read and understand the data received from remote data sources, without having to arrange an explicit data or schema exchange format or being managed by any central authority.

We presented a scenario supported by the DÍGAME architecture, in which data had to be managed across multiple peers in order to accomplish the launch of a new product. Furthermore we described the components of the architecture and discussed its main characteristics.

Further steps include the refinement of the prototype. In addition, we have to examine the impact of DÍGAME on the network traffic, particularly concerning query intensive applications.

Due to its characteristics DÍGAME provides a sophisticated infrastructure for a diversified application field including e-business, e-science, or e-health, initiating the next generation of collaborative work.

Chapter 6

Link Patterns

Collaborative work requires, more than ever, access to data located on multiple autonomous and heterogeneous data sources. The development of these novel information and knowledge platforms, referred to as Semantic Web, information or data grids, and P2P databases, need appropriate modeling and description mechanisms. In this chapter we propose the Link Pattern Catalog as a modeling guideline for recurring problems appearing during the design or description of such information platforms. For this purpose we introduce the Data Link Modeling Language, a language for describing and modeling virtually any kind of data flows in information and knowledge sharing environments. A part of the Link Pattern Catalog presented in this chapter has been published in [PPC04c].

6.1 Introduction

With the rise of filesharing systems like Napster or Gnutella the database community started to seriously adopt the idea of P2P systems to the formerly known loosely coupled databases. While the original systems were only designed to share simple files among a huge amount of peers, we are not restricted to these data sources any more. New developments allow peers to share virtually any data, no matter if it is originated from the Semantic Web, a relational, object-oriented, or XML database. In fact, the data may still come from ordinary flat files.

Apparently we have to deal with a very heterogeneous environment of data sources sharing data, referred to as an information or data grid [CFK+00]. If we allow participants to join or leave information grids at any time (e.g. using P2P concepts [CT04]), we must take a constantly changing constellation of peers into account. Any information grid or knowledge sharing environment built up by these peers can either evolve dynamically or be planned beforehand. In both cases we need a concept in order to describe and understand the interactions among the peers involved. Having such a mechanism, we could not only detect

single data exchanges, but even model and optimize complex data flows of and among the systems.

In this work we adopt commonly used methods for designing data exchanges among peers as *Link Patterns*, suitable especially for information and knowledge sharing environments. Analogous to the intention of the Design Pattern Catalog used for object-oriented software development [GHJV95] we want to provide modeling guidelines for engineers and database designers, engaged in understanding, remodeling, or building up an information grid. Thus information grid architects are provided with a common vocabulary for design and communication purposes.

Up to now data flows in information grids were designed without having a formal background leading to individual solutions for a specific problem. These were only known to a circlet of developers involved into that project. Other designers, engaged with a similar problem would never get in contact with these results and thus make the same mistakes again. Different modeling techniques make it difficult to exchange successfully implemented solutions.

Link Patterns do not claim to introduce novel techniques for sharing, accessing, or processing data in shared environments, but a framework for being able to understand, describe, and model their data flows. They provide a description of basic interactions between data sources and operations on the data exchanged, resulting in a catalog of reusable conceptual units.

A developer may choose Link Patterns to model and describe complex data flows, to identify a single point of failure, or to avoid or consciously insert redundant data exchanges. The composition of Link Patterns is an essential feature of our design method. It gives us the possibility to represent a structured visualization not only of single data linkages, but of the entire information platform.

The remainder of this chapter is organized as follows. In Section 6.2 we introduce DLML, a language for modeling data flows, followed by a structural description of the Link Patterns in Section 6.3. Section 6.4 specifies the Link Pattern Catalog, followed by an example. Section 6.6 catches up some related work and Section 6.7 concludes.

6.2 The Data Link Modeling Language (DLML)

6.2.1 Motivation

The Data Link Modeling Language (DLML) is based on the *Unified Modeling Language* (UML) [GHJV95] notation, but slightly modifies existing components, adds additional elements, and thus extends its functionality. It is a language for modeling, visualizing, and optimizing virtually any kind of data flows in information and knowledge sharing environments.

Modeling: DLML is a language, suitable for modeling, planning, and re-engineering data flows in information and knowledge sharing environments,

e.g. information grids or the Semantic Web, systematically. A Data Link Model built up using this language reflects the logical and not the physical structure of the entire system. It enables the developer to specify the properties and the behavior of existing and novel systems, in order to describe and understand their basic functionalities.

Visualizing: Visualizing data flows is an important assistance in understanding the structure and behavior of an information platform. The impact of ER [Che76] and UML has proven, that a system is easier to grasp and less error-prone, if a graphical visualization technique is provided, which uses a well-defined set of graphical symbols, understood by a broad community. Especially within the analysis of systems with distributed information, it is favorable to have a method, suitable for drawing up a map of relationships between the participating peers, in order to depict global data and knowledge flows.

Validating: The more complex an information or knowledge sharing environment gets, the more error-prone becomes such a system, especially if it was extended throughout the years in numerous steps. Without being carefully validated, such extensions could easily lead to duplicated data items in data grids or conflicts within knowledge sharing environments. Particularly for the Semantic Web, such conflicts could cause semantic reasoners to draw fatal conclusions. Of course a language like the DLML cannot prevent Semantic Web architects from modeling knowledge improperly, but it may help him to identify possible misconfigurations at an early stage.

Optimizing: Besides the modeling, visualization, and validation of existing or planned information sharing environments, DLML can be useful to optimize the whole distributed data management. Redundant data flows and data stocks can systematically be detected and removed, leading to a higher performance of the entire system. Of course, redundancy may explicitly be wanted, in order to achieve a higher fail-safety or a faster access to the data.

Due to the characteristics mentioned above, the Data Link Modeling Language is especially suitable for visualizing data and knowledge flows in distributed information and knowledge sharing environments. It may furthermore be employed to model data management in enterprise information systems, data integration and migration scenarios, or data warehouses, i.e. wherever data has to be accessed across multiple different data sources.

6.2.2 Components

Since DLML is based on UML, its diagrams are constructed in an analogous manner, using a well-defined set of building blocks according to specific rules.

The following components may be used in DLML (Fig. 6.1) to build up a Data Link Model:

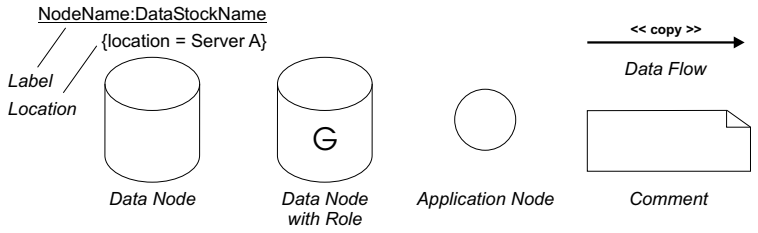

Figure 6.1: DLML components

Nodes: Nodes are data sources, data targets, or applications, usually involved in a data exchange process. They may either be isolated or connected through at least one *data flow*. A data source may be a database (e.g. relational), a flat file (e.g. XML), or something similar, offering data, whereas a data target receives data and stores it locally. An application is a software unit, which accesses or generates data, without maintaining an own physical data stock. Physical data stocks are represented in DLML by *Data Nodes*, applications by *Application Nodes*.

Label: Each node can have a label. It consists of generally two parts separated by a colon: the *node name* and the *data stock name* or *application name* respectively. The data stock name identifies the combination of data and schema information stored at this node. If this data is replicated as an exact and complete copy to another node, the data target has to use the same data stock name. The application is identified by the application name. Analogous to the data stock name, any further instances of the same application have the same application name. In both cases we use the node name to distinguish nodes with the same data stock or application name. Otherwise the node name is optional.

Location: The optional location tagged value specifies the physical location of the node. It either specifies an IP address, a server name, or a room number, helping the developer to locate the Data or Application Node.

Role: A node providing a certain functionality on the data processed, may have a functional role (e.g. filtering or integrating data). This role will usually be implemented as a kind of application, operating directly on the incoming or outgoing data. The name of the role or its abbreviation is placed directly inside the symbol of the node. This information is not only useful for increasing the readability of the model, but also for being able to identify complex relationships.

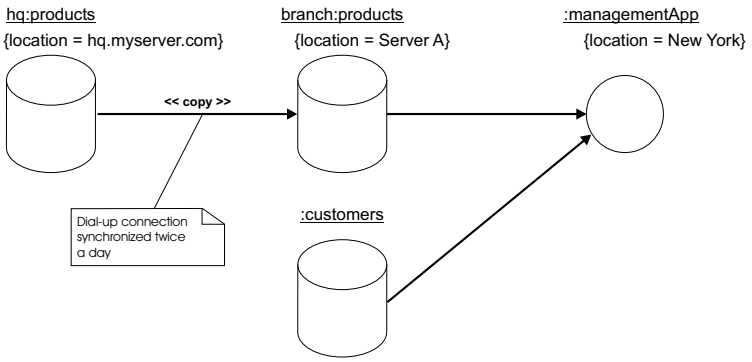

Figure 6.2: DLML example

Data Flow: The data exchange between exactly one data source and one data target is called data flow. The arrow symbolizes the direction, in which data is being sent. A node may have multiple incoming and outgoing data flows. Optionally each data flow may be labeled concerning its behavior, i.e. if the data is being replicated (`<<copy>>`) to the data target or if it is just accessed (`<<access>>`). If data is being synchronized, both data flow arrows may be replaced by one single arrow with two arrowheads.

Comment: A comment may be attached to a component, in order to provide additional information about a node or a data flow. These explanations may concern a node's role, filter criteria, implementation hints, data flow properties, or further annotations important for the comprehension of the model.

6.2.3 Example

We now illustrate the usage of the Data Link Modeling Language with a simplified example. Consider a worldwide operating wholesaler, with an autonomous overseas branch. The headquarters is responsible for maintaining the product catalog (`hq:products`) with its price list, while the customers database (`:customers`) is administrated by the branch itself (Figure 6.2).

The overseas branch is connected to the headquarters by a dial-up connection, not sufficient for accessing the database permanently. For this reason, the product catalog is replicated to the branch twice a day (`branch:products`), where the data may be accessed by the local employees. The branch management uses a special application (`:managementApp`) to access both data stocks in order to generate the annual report for the headquarters.

6.3 Link Patterns

In order to be able to provide a catalog of essential Link Patterns it is necessary to understand what a Link Pattern is. Therefore we present the elements a Link Pattern is composed of, including its name, its classification, or its description. For graphical representation we use the Data Link Modeling Language, specified above.

6.3.1 Elements of a Link Pattern

In this section we present the description of the Link Pattern structure. It is based on the Design Pattern Catalog of Gamma et al. [GHJV95], which has reached great acceptance within the software engineering community. Thus a developer is able to quickly understand and adopt the main concept of each Link Pattern for his own purposes. Each Link Pattern is described by the following elements:

Name: The name of a Link Pattern is its unique identifier. It has to give a first hint on how the pattern should be used. The name is substantial for the communication between or within groups of developers.

Classification: A Link Pattern is classified according to the categories described in Section 6.3.2. The classification organizes existing and future patterns depending on their functionality.

Motivation: Motivating the usage of the pattern is very important, since it explains the developer figuratively the basic functionality. This is done using a small scenario, which illustrates a possible application field of the pattern. Therewith the developer is able to understand and follow the more detailed descriptions in the further sections.

Graphical Representation: The most important part of the pattern description is the graphical representation. It is a DLML diagram and describes the composition and intention of the pattern in an intuitive way. The developer is advised to adopt this representation, wherever he has identified the related functionality in his own information or knowledge sharing environment.

Description: The composition of the Link Pattern is described in-depth in this section, including every single component and its detailed functionality. The explanation of the local operations on each node and data flows between the components involved, points up the intended functionality of the whole pattern described. This description shall give the user both, a guidance through the identification process and instructions for its proper usage.

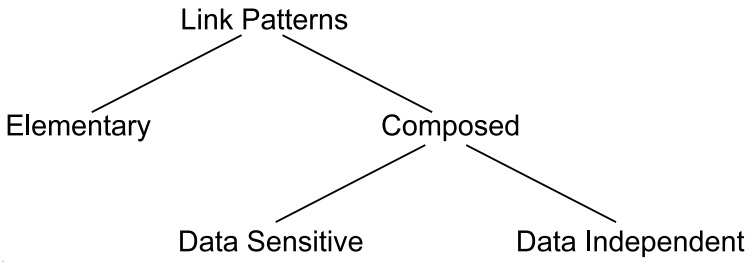

Figure 6.3: Link Pattern Catalog Classification

Challenges: Besides the general instructions given in the prior section, this section shall give hints for sources of error in the implementation process of this pattern. The developer shall get ideas, of how to identify and avoid pitfalls, arising in a certain context (e.g. interaction with other Link Patterns). This part of a Link Pattern corresponds approximately to the Implementation section of a Design Pattern (cf. [GHJV95]).

6.3.2 Classification

A classification of the Link Pattern Catalog shall provide an organized access to all Link Patterns presented. Patterns situated in the same class have similar structural or functional properties, depending on the complexity of their implementation. Although a categorization of a very limited number of patterns may seem superfluous, we have decided to include this into our Link Pattern Catalog, since it shall help developers to allocate and evaluate the pattern required. Furthermore it should stimulate the developer to find and rate novel patterns, not yet included in the catalog.

Figure 6.3 depicts the classification of our Link Pattern Catalog we have chosen. The patterns presented can be divided into two main categories, *Elementary Link Patterns* and *Composed Link Patterns*. In fact this classification is not completed, but shall provide a starting point for further extension.

Elementary Link Pattern: An Elementary Link Pattern is the smallest unit for building up an information grid model. It consists of exclusively one single node and at least one data flow connected to it. Each Data Link Model is composed of several Elementary Link Patterns, linked together with data flows in an appropriate way. Please note, that a single Elementary Link Pattern is not yet a reasonable Data Link Model, since any data flow must have at least one node offering data and one node receiving data.

Elementary Link Patterns are easy to understand and easy to implement, since they concern only a single node, a small set of data flows, and do not include basically any data processing logic. It must be pointed out,

that the Elementary Link Patterns consist only of two main patterns, the *Basic Data Node* and the *Basic Application Node*, and its derivatives (e.g. Publisher and Generator, discussed in Section 6.4).

Composed Link Pattern: Composed Link Patterns are built up by combining at least two Elementary Link patterns in a specific way, in order to realize a particular functionality. A Composed Link Pattern may hereby be composed out of both, Elementary or other Composed Link Patterns. A pattern has to represent a prototype or solution for a recurring sort of problem. Please keep in mind, that an arbitrary combination of different patterns will not automatically lead to a reasonable Composed Link Pattern.

In contrast to the Elementary Link Patterns, we have to deal in this context with a more complex kind of patterns. They do not only include more nodes, but may even represent a quite sophisticated way of linking them. Besides, each node may additionally process the data received or sent. The fact, that it may act differently depending on the data involved, is an essential property of Composed Link Patterns and justifies the creation of two subclasses:

Data Sensitive Link Pattern: As soon as a node included in a Composed Link Pattern acts depending on the data it processes, the entire pattern is called a Data Sensitive Link Pattern. This data processing logic implemented on such a node may depend on and be applied to incoming and/or outgoing data. The operations of this application can either create, alter, or filter data.

Data Independent Link Pattern: Any Composed Link Pattern, not classified as Data Sensitive, belongs to this class. In contrast to the patterns described above, data is not being modified, but sent or received as is. A rather crucial topic is the topology of the nodes and data flows involved, which is most relevant for the creation and functionality of this kind of patterns.

6.3.3 Usage

This section describes how Link Patterns can be useful to develop, maintain, analyze, or optimize both, straightforward and complex data flows in information sharing environments. There are basically two methods, how Link Patterns can improve the work of developers:

Analyzing existing systems: Many existing information or knowledge sharing environments have arisen during the years without being planned centrally or consistently. Even if they were planned initially, they usually tend to

spread in an uncontrolled way. In such an environment it is vital to have supporting tools, helping to understand and later optimize an existing system.

First of all a map or model of the existing system has to be created, e.g. with DLML presented in Section 6.2. Afterwards we examine successively smaller parts of the model, in order to match them to existing Link Patterns of the Catalog. As a result we get a revised model containing basic information on the composition and functionality of subsystems, including their data processing and data flows. With this information in mind, we are now able to derive information on data flows and interaction of nodes inside the Data Link Model. This enables us to perform optimizations like detecting and eliminating vulnerabilities or handling redundancies.

Link Patterns may thus not replace human expertise for understanding existing information grids, but give support in the process of recognizing global data flows and therewith interpret the purpose of the entire system.

Composing new models: As already mentioned a Link Pattern may not only improve the process of understanding an existing model, but is also a support for modeling new systems. An architect needs to have a clear idea of what the system should do. Depending on the data sources available, the local requirements on the nodes, and the results he wants to achieve, he can combine nodes and data flows, according to Link Patterns, until the entire system realizes the intended functionality. Link Patterns hereby guarantee a common language, understood by other developers, not yet involved in the modeling. Each developer is thus able to quickly get a general idea of the system modeled at any time. Furthermore they accelerate the development process, since they provide well tried solutions for recurring problems, leading to an efficent system of high quality.

6.4 Link Pattern Catalog

In this section we give an introduction to the Link Pattern Catalog. This includes a graphical overview over the main Link Patterns in DLML, as well as a detailed description of selected patterns. As mentioned beforehand the Link Patterns can be classified according to the classification presented in Section 6.3.2. Since any Composed Link Pattern either belongs to the Data Sensitive or to the Data Independent Link Patterns, we organize the catalog as follows:

6.4.1 Elementary Link Patterns

The Elementary Link Patterns are the basic building blocks of a Data Link Model. They consist of the two basic patterns, described below, and its derivatives. All Elementary Link Patterns are depicted in Figure 6.4.

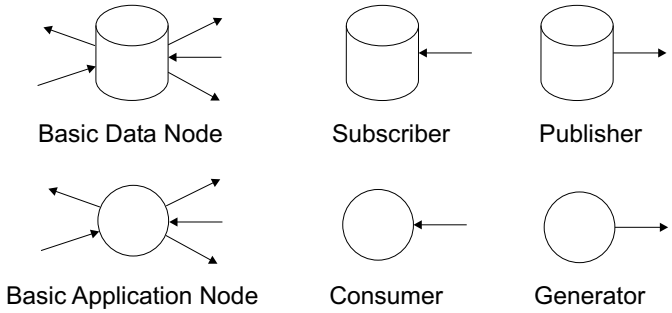

Figure 6.4: Elementary Link Patterns

Basic Data Node

Classification: Elementary Link Pattern

Motivation: This pattern is one of the basic building blocks of a Data Link Model. Each incoming or outgoing data flow of a Data Node is modeled using this Link Pattern.

Graphical Representation: See Figure 6.4

Description: A Basic Data Node is a DLML Data Node, which receives data through incoming data flows, stores it locally, and simultaneously propagates data, held in its own data stock. If a Basic Data Node does only have outgoing or incoming data flows, it applies the *Publisher Pattern* or the *Subscriber Pattern* respectively. If it does neither have any incoming, nor any outgoing data flows, the Data Node is called *isolated*.

Challenges: One of the main challenges to take in this pattern is the proper coordination of incoming and outgoing data flows. At first all incoming data has to be stored permanently on the local data stock, without violating any constraints, before it may be propagated again to other nodes.

Basic Application Node

Classification: Elementary Link Pattern

Motivation: This pattern is one of the basic building blocks of a Data Link Model. All applications, relevant for a Data Link Model, are based on this pattern.

Graphical Representation: See Figure 6.4

Description: An application interacting with arbitrary Data or Application Nodes, is represented by this pattern. The application does not only receive, but also propagate data. If a Basic Application Node does only have outgoing or incoming data flows, it applies the *Generator Pattern* or the *Consumer Pattern* respectively. If it does neither have any incoming nor any outgoing data flows, the Application Node is called *isolated*.

Challenges: Propagated data can either be received or generated. All data manipulations on incoming data, which have to be propagated, have to be processed in real-time, without storing data locally.

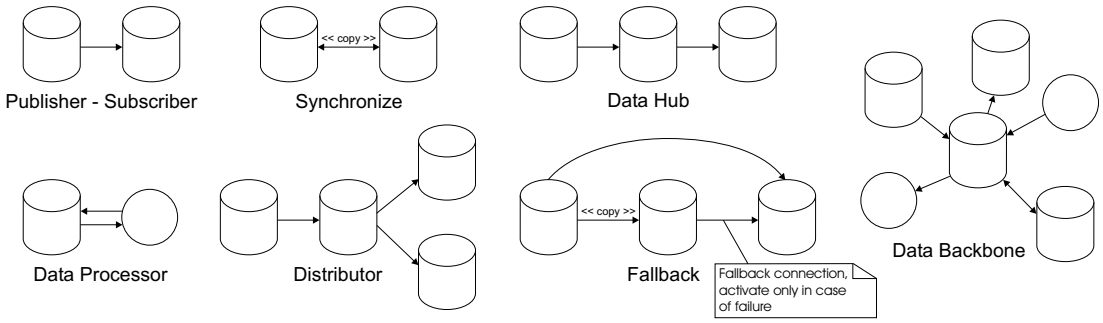

Figure 6.5: Data Independent Link Patterns

6.4.2 Data Independent Link Patterns

The Data Independent Link Patterns belong to the Composed Link Patterns. These patterns describe a functionality, which only depends on their structure, i.e. the way nodes and data flows are combined. A graphical overview of the patterns in this class is given in Figure 6.5, of which the *Data Backbone* is described exemplarily.

Data Backbone

Classification: Data Independent Link Pattern

Motivation: A Data Backbone is used, wherever a centralization of data sharing or data access has to be realized. This is typically required, if data stocks are re-centralized, a central authority wants to keep track on all data flows, or data exchanges have to be established among multiple data stocks and applications.

Graphical Representation: See Figure 6.5

Description: The Data Backbone Pattern consists of several nodes, linked together in a specific way. A designated node, called Data Backbone, is either data source or data target for all data flows in this pattern. All nodes, including the Data Backbone itself, can be data stocks or applications. Data is always propagated from data sources to the Data Backbone, where it may be accessed or propagated once again to other target nodes. Direct data flows between nodes, which are not the Data Backbone, are avoided.

Challenges: Since the Data Backbone is involved in all data flows, it has a crucial position in this part of the information grid. Thus, a Data Backbone node has to provide a high quality of service, concerning disk space, network connection, and processing performance. If the quality of service required cannot be provided, the Data Backbone may easily become a bottleneck. Furthermore a breakdown of this node could lead to a collapse of the entire data sharing infrastructure, which makes it to a single point of failure.

6.4.3 Data Sensitive Link Patterns

Contrary to the Data Independent Link Patterns, the patterns described in this section are not only classified according to their structural properties, but particularly because of their data processing functionality. A graphical representation of these Data Sensitive Link Patterns can be found in Figure 6.6, while a detailed description is only given for the *Gatekeeper* and *Semantic Translation Patterns*.

Gatekeeper

Classification: Data Sensitive Link Pattern

Motivation: A Gatekeeper is used to control data flows according to specific rules (e.g. Access Control Lists), stored separately from the data processed. It is responsible for providing the target nodes with the accessible data required. The application of this pattern is not limited to data security matters. It may actually be applied to any node, which has to supply different target nodes with specific (e.g. manipulated or filtered) data flows.

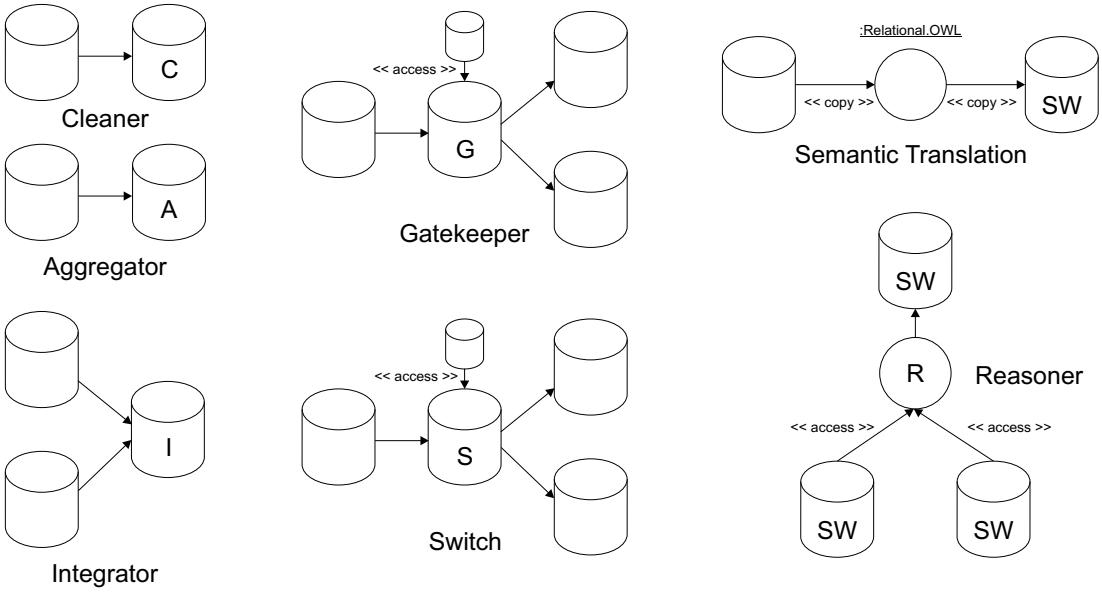

Figure 6.6: Data Sensitive Link Patterns

Graphical Representation: See Figure 6.6

Description: A Gatekeeper is a designated node, which distributes data according to specific rules, eventually stored separately. Local or incoming data of a Gatekeeper is accessed by target nodes. Before this access can be admitted, the Gatekeeper has to check the permissions. Thus, corresponding to the rules processed, neither all data stored in the Gatekeeper, nor all data requested by the target nodes has to be transmitted.

Challenges: The rules and techniques, which are used by the Gatekeeper in order to secure access to the data, have to be robust and safe. The Gatekeeper needs a mechanism to identify and authenticate the source and target nodes (e.g. IP address, public key, username and password, or identifiers as in Chapter 4), which may be stored in a separated data stock. Due to its vital position in the exchange process, this information has to be protected from unauthorized access. The Gatekeeper must be able to rely on the correctness, authenticity, and availability of the rules required.

Semantic Translation

Classification: Data Sensitive Link Pattern

Motivation: Whenever data originally stored in a relational database has to be made accessible for Semantic Web applications, a Semantic Translation should be made. It automatically transforms relational legacy data into a

semantic rich representation, processable by most Semantic Web application using its own built-in functionality. This data can then be used for further data processing or reasoning tasks and may be mapped to a target ontology (cf. Section 3.3).

Graphical Representation: See Figure 6.6

Description: A Semantic Translation consists of three nodes, two DLML Data Nodes and one DLML Application Node. The latter is an instance of the Relational.OWL application (cf. Section 5.1), which retrieves data stored in a relational database, transforms it into its Relational.OWL representation (cf. Section 3.3.2) and makes it part of the Semantic Web. The data flows from the relational database to the Relational.OWL application and to the DLML Data Node tagged with `SW`, i.e. the *Semantic Web* role, are both marked with `<<copy>>` labels, since the data is copied out of the relational database into the Semantic Web.

Challenges: Implementing a Semantic Translation should not cause serious problems, since it is performed automatically by the Relational.OWL application. Nevertheless, there are some precautions to be considered.

Since a Semantic Translation does not result in Semantic Web objects containing real semantics, but in data items represented within table and column objects, some applications needing to perform reasoning tasks, may not be able to interpret this data correctly. In this case, the Relational.OWL representation of the database has to be mapped additionally to a target ontology (cf. Section 3.3).

Moreover, it has to be considered that a Semantic Translation implemented following this Link Pattern is performed using the Relational.OWL application. As a consequence, the data provided to the Semantic Web corresponds to a snapshot of the original relational database, i.e. it may not always be up-to-date. If the information in the original relational database is time-critical, a version of the Semantic Translation pattern should be regarded, where the data transformation is performed in real-time by an application like RDQuery (cf. Section 5.2).

6.5 Example

This section provides an example of how to model a new information sharing environment of a worldwide operating company. The headquarters of the company are located in New York. It has additionally branches in Düsseldorf (head office of the European branches), Paris, Bangalore, and Hong Kong. Each branch maintains its own database containing sales figures, collected by local applications. For backup and subsequent data analysis, this data has to be replicated

to the headquarters. Additionally, the Düsseldorf branch needs to be informed about the ongoing sales activities of the Paris branch. To simplify the centralized backup, the company has decided to forbid any data exchanges between the single branches.

The central component of this infrastructure is the backup system in New York. It collects the sales data from all branches, without integrating them. Additionally it provides the Düsseldorf branch with all the information required from Paris. Since the headquarters in New York want to analyze the entire data stock of the company, a data warehouse, based on the data of the backup system, is set up. Having a certain local autonomy, the data provided by the European branches and the remaining branches have some structural differences. For this reason, the data has to be integrated prior to the aggregation required for the data warehousing analysis.

Using the Link Patterns proposed, we are now able to model the enterprise information grid as depicted in Figure 6.7.

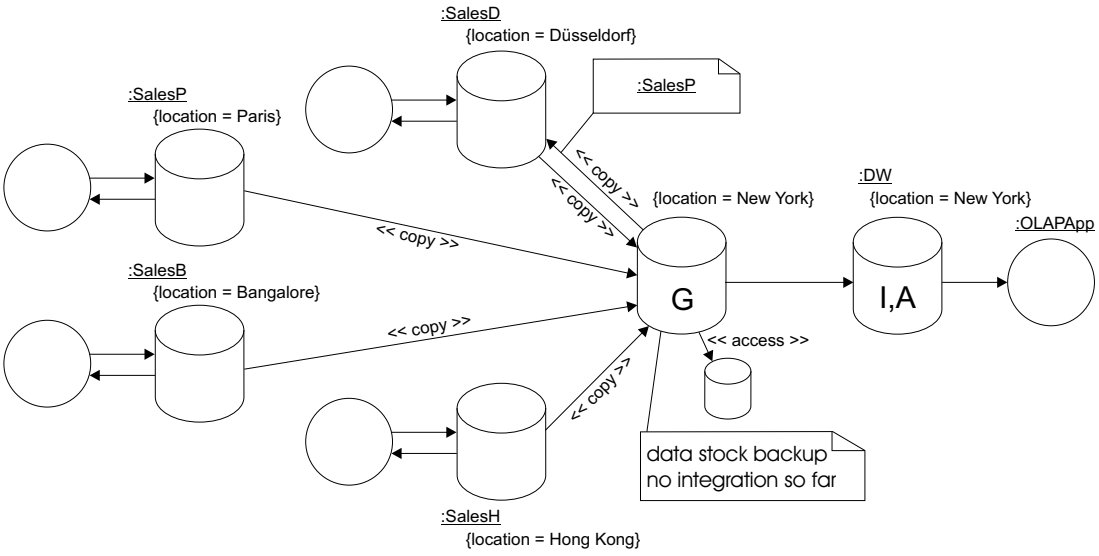

Figure 6.7: Example using Link Patterns

The local applications, which maintain the local sales databases, are modeled using the *Data Processor*. This data is replicated to the backup system in New York, realized as a *Gatekeeper*. It thus controls the data flows from the branches to the data warehouse and to the Düsseldorf branch. It must be guaranteed, that the data targets get only their designated data, i.e. neither data from Bangalore, nor from Hong Kong is accessible for the European head office in Düsseldorf. The data warehouse is realized by a node, which integrates several data sources using common integration strategies (*Integrator Pattern*) and aggregates the data afterwards (*Aggregator Pattern*), in order to provide OLAP applications with a homogenous data stock.

Please keep in mind, that the Data Link Model presented in Figure 6.7 reflects the logical structure of the information platform, not the physical. This means, that the nodes of the model do not have to be located on different machines.

6.6 Related Work

Data Flow analysis and modeling has been a focus of researchers for decades. Earlier work concentrates mainly on data flows in computer architectures and software components (e.g. [Win78, CPRZ89]). Later on, data flows were also used for query processing and optimization in database systems. For instance, Teeuw and Blanken [TB93] compare control versus data flow mechanisms controlling the execution of database queries on parallel database systems.

Dennis and Misunas present in [DM75] a Basic Data-Flow language, which expresses graphically the data dependencies within a program. In this data flow graph model, instructions are represented by nodes and paths stand for data or control flows. Although this language was originally designed for software development, it may be seen as an early forerunner, in designing data flows among different data sources. A specialized data flow graph is introduced by Eich and Wells [EW88], which can be used for scheduling database queries within multiprocessor environments or databases distributed over a network [BH89]. Thus, both approaches apply data flow concepts to database processing.

The Link Patterns are tightly coupled to the Design Patterns of the object-oriented software design [GHJV95, BMR+96] and Enterprise Application Integration (EAI) [HW03], since they represent prototypes or solutions for recurring problems. Contrary to these patterns, Link Patterns are not intended to solve recurring problems in software design or EAI, but to provide modeling and description guidelines for information grids, focusing exclusively on data flows.

As a possible application field of our Link Patterns we suggest modeling or visualizing information grids, i.e. heterogeneous environment of data sources sharing data, or modern information infrastructures, based on P2P concepts (e.g. [HIMT03] or Section 5.3).

6.7 Discussion and Future Work

In this chapter we have presented Link Patterns as guidelines for modeling and describing data flows between nodes in information and knowledge sharing environments. The Link Pattern Catalog consists of prototypes or solutions for recurring problems and therewith supports developers to model, describe, and understand complex information grids. Furthermore the Link Patterns provide a common vocabulary for design and communication purposes, enabling developers to exchange successfully implemented solutions.

Additionally we have introduced the Data Link Modeling Language (DLML) for modeling, visualizing, and optimizing data flows, especially suitable for knowledge and information grids. This language based on UML consists of a well-defined set of building blocks, representing data nodes, application nodes and data flows between them. They can be combined according to specific rules, to build up the Data Link Model of an information sharing environment.

The concepts we have presented in this chapter are ideal to generate a static model of data and application nodes with their corresponding data flows. In future work we have to consider dynamically changing and evolving environments, in which nodes constantly join or leave the grid. This may not only affect the Link Pattern Catalog, but also the Data Link Modeling Language. Furthermore the Catalog has to be enhanced, in order to include novel Link Patterns, not yet identified. The entire Link Pattern Catalog shall provide developers with an extensive reference guideline for modeling information sharing environments.

Chapter 7

Conclusion

The vision of the Semantic Web [BLHL01] is, despite all the efforts to build up the next generation Web, still more dream than reality. The main reason for this issue is the lack of data, since the vast majority of information is still stored in (relational) databases and thus unavailable for most Semantic Web applications.

In this thesis we have introduced Relational.OWL, a technique to automatically transform data and schema components of a relational database into a Semantic Web representation. First, the schema of the corresponding database is extracted based on the Relational.OWL ontology. This schema representation itself is then used as a new ontology to describe the data items stored in that specific database.

Despite being processable by any application understanding RDF and OWL, the data extracted using Relational.OWL still lacks *real* semantic meaning, since the information originally stored in relational tables is represented within a table object and not within appropriate Semantic Web objects. Since many applications performing reasoning tasks require the data to be represented as *real* Semantic Web objects, we have described how to map the Relational.OWL representation of a database to a specific target ontology using an RDF query language.

Furthermore, we have introduced the novel URI scheme `db` for identifying not only databases, but also their schema and data components like tables or columns. Particularly suitable for the Semantic Web, this URI enables us to backtrack each relational data item transformed into a Semantic Web object to its original location in the relational database.

Finally, we have presented the Link Pattern Catalog, which consists of prototypes or solutions for recurring problems and therewith supports developers to model, describe, and understand complex information sharing environments like the Semantic Web. Additionally, Link Patterns provide a common vocabulary for design and communication purposes, enabling developers to exchange successfully implemented solutions.

With the techniques presented in this thesis, relational databases and the Se-

mantic Web have approached significantly. We are now able to map relational data automatically into the Semantic Web, enabling Semantic Web applications to process formerly relational data using their own built-in functionality. Since some applications require the data items to be mapped to a specific target ontology, instead of using the automatically generated data representation, we have shown how to perform such a mapping task using a suitable RDF query language. Using our novel URI, we are additionally able to unambiguously identify each data item and backtrack it to its original storage location in the relational database.

Having implemented a relational to Semantic Web solution successfully, a user is now able to make his solution available to other users confronted with a similar problem. Using a Link Pattern, he shares successfully implemented solutions and hence helps other users to realize their contribution to the Semantic Web.

As a consequence, relational databases and the Semantic Web are no longer isolated data banks lacking of any common data flows, but each relational database can now be regarded as an integral part of the Semantic Web.

Bibliography

[ABM05] Yuan An, Alexander Borgida, and John Mylopoulos. Inferring Complex Semantic Mappings Between Relational Tables and Ontologies from Simple Correspondences. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part II*, volume 3761 of *Lecture Notes in Computer Science*, pages 1152–1169. Springer Verlag, 2005.

[AFZ96] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Disseminating Updates on Broadcast Disks. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 354–365. Morgan Kaufmann, 1996.

[AFZ97] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 183–194. ACM Press, 1997.

[Av03] Grigoris Antoniou and Frank van Harmelen. Web Ontology Language: OWL. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies in Information Systems*, International Handbooks on Information Systems. Springer Verlag, 2003.

[Av04] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.

[Bac69] Charles W. Bachman. Data structure diagrams. *DATA BASE*, 1(2):4–10, 1969.

[Bar99] Daniel Barbará. Mobile Computing and Databases - A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[BBFS05] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In *Reasoning Web, First International Summer School*, volume 3564 of *Lecture Notes in Computer Science*, pages 35–133. Springer Verlag, 2005.

[BCF+05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/2005/CR-xquery-20051103/`, 2005. W3C Candidate Recommendation.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[Be004] RDF/XML Syntax Specification (Revised). `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`, February 2004. W3C Recommendation.

[Ber00] Michael K. Bergman. The Deep Web: Surfacing Hidden Value. BrightPlanet LLC, 2000. White paper.

[BGK+02] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu:. Data Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the Fifth International Workshop on the Web and Databases, WebDB 2002*, Madison, WI, 2002.

[BGM04] Dan Brickley, Ramanathan V. Guha, and Brian McBride. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`, February 2004. W3C Recommendation.

[BH89] Lubomir Bic and Robert L. Hartmann. AGM: A Dataflow Database Machine. *ACM Transactions on Database Systems (TODS)*, 14(1):114–146, 1989.

[BI94] Daniel Barbará and Tomasz Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 1–12. ACM Press, 1994.

[BIPG92] José M. Blanco, Arantza Illarramendi, José M. Pérez, and Alfredo Goñi. Making a Federated Database System Active. In *Database and*

Expert Systems Applications (DEXA 1992), pages 345–351. Springer Verlag, 1992.

[Biz03] Christian Bizer. D2R MAP-A Database to RDF Mapping Language. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003 - Posters*, 2003.

[BJY+02] Tom Barrett, David Jones, Jun Yuan, John Sawaya, Mike Uschold, Tom Adams, and Deborah Folger. RDF Representation of Metadata for Semantic Integration of Corporate Information Resources. In *International Workshop Real World and Semantic Web Applications 2002*, 2002.

[BKD+01] Jeen Broekstra, Michel Klein, Stefan Decker, Dieter Fensel, Frank van Harmelen, and Ian Horrocks. Enabling Knowledge Representation on the Web by Extending RDF Schema. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China, May 1-5, 2001*, pages 467–478. ACM Press, 2001.

[BL98a] Tim Berners-Lee. Relational Databases and the Semantic Web (in Design Issues). `http://www.w3.org/DesignIssues/RDB-RDF.html`, September 1998.

[BL98b] Tim Berners-Lee. Web design issues; What a semantic can represent. `http://www.w3.org/DesignIssues/RDFnot.html`, September 1998.

[Bla98] Kenneth R. Blackman. Technical Note - IMS Celebrates Thirty Years as an IBM Product. *IBM Systems Journal*, 37(4):596–603, 1998.

[BLFM98] Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, 1998.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[BLM02] Tim Berners-Lee and Eric Miller. The Semantic Web lifts off. *ERCIM News*, 51:9–11, October 2002.

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`, 2004. W3C Recommendation.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., 1996.

[BOF+04] Christopher Brewster, Kieron O'Hara, Steve Fuller, Yorick Wilks, Enrico Franconi, Mark A. Musen, Jeremy Ellman, and Simon Buckingham Shum. Knowledge Representation with Ontologies: The Present and Future. *IEEE Intelligent Systems*, 19(1):72–81, 2004.

[BPSM04] Tim Bray, Jean Paoli, and Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/2004/REC-xml-20040204`, 2004. W3C Recommendation.

[BS02] François Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Erfurt, Germany, October 7-10, 2002, Revised Papers*, Lecture Notes in Computer Science, pages 295–310. Springer Verlag, 2002.

[BS04] Christian Bizer and Andy Seaborne. D2RQ -Treating Non-RDF Databases as Virtual RDF Graphs. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004.*, 2004. poster presentation.

[BvH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, and Franklin W. Olin. OWL Web Ontology Language Reference. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`, 2004. W3C Recommendation.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In Randall Rustin, editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, pages 249–264. ACM, 1974.

[CDES05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1216–1227. VLDB Endowment, 2005.

[CDF+98] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom M. Mitchell, Kamal Nigam, and Seán Slattery. Learning to

extract symbolic knowledge from the World Wide Web. In *Proceedings of AAAI-98, 15th Conference of the American Association for Artificial Intelligence*, pages 509–516. AAAI Press, Menlo Park, US, 1998.

[CFK+00] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[CG01] Bengt Carlsson and Rune Gustavsson. The Rise and Fall of Napster - An Evolutionary Approach. In *Active Media Technology, 6th International Computer Science Conference, AMT 2001, Hong Kong, China, December 18-20, 2001, Proceedings*, volume 2252 of *Lecture Notes in Computer Science*, pages 347–354. Springer Verlag, 2001.

[CGMW96] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. A Toolkit For Constraint Management In Heterogeneous Information Systems. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 56–65, 1996.

[CH79] Ashok K. Chandra and David Harel. Computable Queries for Relational Data Bases (Preliminary Report). In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing, 30 April-2 May, 1979, Atlanta, Georgia, USA*, pages 309–318. ACM Press, 1979.

[Cha76] Donald D. Chamberlin. Relational Data-Base Management Systems. *ACM Computing Surveys (CSUR)*, 8(1):43–66, 1976.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod72] Edgar F. Codd. Relational Completeness of Data Base Sublanguages. *R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[Cod79] Edgar F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

[Con97] Stefan Conrad. *Föderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer Verlag, Berlin, 1997.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[CT04] Mario Cannataro and Domenico Talia. Semantics and Knowledge Grids: Building the Next-Generation Grid. *IEEE Intelligent Systems*, 19(1):56–63, 2004.

[Cyg05] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Labs, Bristol, UK, 2005.

[Dat82] Christopher J. Date. A Formal Definition of the Relational Model. *SIGMOD Record*, 13(1):18–29, 1982.

[Dat00] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, 2000.

[DBJS03] David De Roure, Mark A. Baker, Nicholas R. Jennings, and Nigel R. Shadbolt. The Evolution of the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. John Wiley & Sons Inc., New York, April 2003.

[DM75] Jack B. Dennis and David P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pages 126–132. ACM Press, 1975.

[DMDH02] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to Map between Ontologies on the Semantic Web. In *Proceedings of the Eleventh International World Wide Web Conference, WWW2002, Honolulu, Hawaii, USA, 7-11 May 2002*, pages 662–673. ACM Press, 2002.

[DMDH04] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Y. Halevy. Ontology Matching: A Machine Learning Approach. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 385–404. Springer Verlag, 2004.

[DRR+06] Olivier Dameron, Elodie Roques, Daniel Rubin, Gwenaelle Marquet, and Anita Burgun. Grading Lung Tumors using OWL-DL based reasoning . In *Proceedings of the 9th International Protege Conference, Stanford USA*, 2006.

[DS04] Mike Dean and Guus Schreiber. OWL Web Ontology Language Reference. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`, 2004. W3C Recommendation.

[Dum01] Edd Dumbill. The semantic web: A primer. `http://www.xml.com/pub/a/2000/11/01/semanticweb/`, November 2001.

[EM02] Andrew Eisenberg and Jim Melton. SQL/XML is Making Good Progress. *SIGMOD Record*, 31(2):101–108, 2002.

[EM04] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML. *SIGMOD Record*, 33(3):79–86, 2004.

[EW88] Margaret H. Eich and David L. Wells. Database Concurrency Control Using Data Flow Graphs. *ACM Transactions on Database Systems (TODS)*, 13(2):197–227, 1988.

[FGM+99] Roy Thomas Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. RFC 2616: Hypertext Transfer Protocol - HTTP/1.1, 1999.

[FHVB04] Flavius Frasincar, Geert-Jan Houben, Richard Vdovjak, and Peter Barna. RAL: An Algebra for Querying RDF. *World Wide Web*, 7(1):83–109, 2004.

[Fre98] Dayne Freitag. Information Extraction from HTML: Application of a General Machine Learning Approach. In *Proceedings of AAAI-98, 15th Conference of the American Association for Artificial Intelligence*, pages 517–523, 1998.

[GHI+01] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan Suciu. What Can Databases Do for Peer-to-Peer? In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB '2001), Santa Barbara, CA*, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, Montreal, Canada, 1996. ACM Press.

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2002.

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[GW93] Ashish Gupta and Jennifer Widom. Local Verification of Global Integrity Constraints in Distributed Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 49–58, Washington, DC, 1993.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[Han98] David J. Hand. Data Mining: Statistics and More? *The American Statistician*, 52(2):112–118, 1998.

[Har02] Takahiro Hara. Cooperative caching by mobile clients in push-based information systems. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 186–193, 2002.

[HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 502–517, 2004.

[HdCD+05] Frank W. Hartel, Sherri de Coronado, Robert Dionne, Gilberto Fragoso, and Jennifer Golbeck. Modeling a description logic vocabulary for cancer research. *Journal of Biomedical Informatics*, 38(2):114–129, 2005.

[Hef04] Jeff Heflin. OWL Web Ontology Language Use Cases and Requirements. `http://www.w3.org/TR/2004/REC-webont-req-20040210/`, 2004.

[HEPS02] Masahiro Hori, Jérôme Euzenat, and Peter F. Patel-Schneider. OWL Web Ontology Language XML Presentation Syntax. `http://www.w3.org/TR/owl-xmlsyntax/`, 2002. W3C Recommendation.

[HIMT03] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In *Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003.*, pages 556–567, 2003.

[HJKS06] Katja Hose, Andreas Job, Marcel Karnstedt, and Kai-Uwe Sattler. An Extensible, Distributed Simulation Environment for Peer Data Management Systems. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 1198–1202. Springer Verlag, 2006.

[HM85] Dennis Heimbigner and Dennis McLeod. A Federated Architecture for Information Management. *ACM Transactions on Information Systems (TOIS)*, 3(3):253–278, 1985.

[Hos02] Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery. In *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, pages 126–144, 2002.

[HS05] Stephen Harris and Nigel Shadbolt. SPARQL Query Processing with Conventional Relational Database Systems. In *Web Information Systems Engineering - WISE 2005 Workshops, WISE 2005 International Workshops, New York, NY, USA, November 20-22, 2005, Proceedings*, volume 3807 of *Lecture Notes in Computer Science*, pages 235–244. Springer Verlag, 2005.

[HSWW03] L. Hollink, A. Th. Schreiber, B. Wielemaker, and B. Wielinga. Semantic Annotation of Image Collections. In *Proceedings of the KCAP'03 Workshop on Knowledge Markup and Semantic Annotation, Florida, USA, October 2003.*, 2003.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[Ian01] Renato Iannella. Representing vCard Objects in RDF/XML. `http://www.w3.org/TR/vcard-rdf`, 2001. W3C Note.

[Int88] International Organization for Standardization. *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times*. Geneva, Switzerland, 1988. See also 1-page correction, ISO 8601:1988/Cor 1:1991.

[Int98] International Organization for Standardization. *ISO/IEC 6523-1:1998 Information technology – Structure for the identification of organizations and organization parts – Part 1: Identification of organization identification schemes*. Geneva, Switzerland, 1998.

[Int01] International Organization for Standardization. *ISO/IEC 9075-9 : 2001: Title: Information technology – Database languages – SQL–Part 9: Management of External Data (SQL/MED)*. Geneva, Switzerland, 2001.

[Int02] International Telecommunication Union. *Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU-T Recommendation X.680*. Geneva, Switzerland, 2002.

[Int03a] International Organization for Standardization. *ISO/IEC 13250 : 2003: Title: Information technology – SGML applications – Topic maps)*. Geneva, Switzerland, 2003.

[Int03b] International Organization for Standardization. *ISO/IEC 9075 : 2003: Title: Information technology – Database languages – SQL*. Geneva, Switzerland, 2003.

[Int03c] International Organization for Standardization. *ISO/IEC 9075-14 : 2003: Title: Information technology – Database languages – SQL–14: XML-Related Specifications (SQL/XML)*. Geneva, Switzerland, 2003.

[Int03d] International Organization for Standardization. *ISO/IEC 9075-9 : 2003: Title: Information technology – Database languages – SQL–Part 9: Management of External Data (SQL/MED)*. Geneva, Switzerland, 2003.

[Int03e] Internet Assigned Numbers Authority. Uniform Resource Identifier (URI) SCHEMES. `http://www.iana.org/assignments/uri-schemes`, 2003.

[Jac05] Ian Jacobs. World Wide Web Consortium Process Document. `http://www.w3.org/2005/10/Process-20051014/`, 2005.

[Jav06] JavaCC - Java Compiler Compiler. `https://javacc.dev.java.net/`, 2006.

[JDB06] JDBC Technology. `http://java.sun.com/products/jdbc/`, 2006.

[Jen06] Jena - A Semantic Web Framework for Java. `http://jena.sourceforge.net/`, 2006.

[JT75] Paul R. Johnson and Robert H. Thomas. RFC 677: Maintenance of duplicate databases, 1975.

[KAM03] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 325–336. ACM Press, 2003.

[KCGS95] Won Kim, Injun Choi, Sunit Gala, and Mark Scheevel. On Resolving Schematic Heterogeneity in Multidatabase Systems. In W. Kim, editor, *Modern Database Systems*, chapter 26, pages 521–550. ACM Press, New York, NY, 1995.

[KCPA01] Gregory Karvounarakis, Vassilis Christophides, Dimitris Plexousakis, and Sofia Alexaki. Querying RDF Descriptions for Community Web Portals. In *17èmes Journées Bases de Données Avancées, BDA'2001, Agadir, Maroc*, pages 133–144, 2001.

[KlC04] Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, February 2004. W3C Recommendation.

[KMH06] Vipul Kashyapa, Alfredo Morales, and Tonya Hongsermeiera. On Implementing Clinical Decision Support: Achieving Scalability and Maintainability by Combining Business Rules and Ontologies. Technical Report CIRD-20060322-01, Clinical Informatics R&D, Partners HealthCare System, Wellesley, MA, 2006.

[LA86] Witold Litwin and Abdelaziz Abdellatif. Multidatabase Interoperability. *Computer*, 19(12):10–18, 1986.

[Las97] Ora Lassila. Introduction to RDF Metadata. `http://www.w3.org/TR/NOTE-rdf-simple-intro-971113.html`, November 1997. W3C NOTE 1997-11-13.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[LL95] Stefan M. Lang and Peter C. Lockemann. *Datenbankeinsatz.* Springer Verlag, 1995.

[LS99] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`, February 1999. W3C Recommendation.

[Mai83] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[Mat06] Michael Matuschek. Extraktion relationaler Daten für das Semantic Web in Oracle 10g. Bachelor's Thesis, Heinrich-Heine-Universität Düsseldorf, 2006.

[Mea00] Michael Mealling. RFC 3001: A URN Namespace of Object Identifiers, 2000.

[Mel99] Sergey Melnik. Algebraic Specification for RDF Models. `http://www-diglib.stanford.edu/diglib/ginf/WD/rdf-alg/rdf-alg.pdf`, 1999. Working Draft.

[Mel01] Sergey Melnik. Storing RDF in a Relational Database. `http://www-db.stanford.edu/~melnik/rdf/db.html`, 2001.

[MFHS02] Deborah L. McGuinness, Richard Fikes, James A. Hendler, and Lynn Andrea Stein. IEEE Intelligent Systems: DAML+OIL: An Ontology Language for the Semantic Web. *IEEE Distributed Systems Online*, 3(11), 2002.

[MM04] Frank Manola and Eric Miller. RDF primer. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`, 2004. W3C Recommendation.

[MMJ+01a] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna G. Kulkarni, Peter M. Schwarz, and Kathy Zeidenstein. SQL and Management of External Data. *SIGMOD Record*, 30(1):70–77, 2001.

[MMJ+01b] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna G. Kulkarni, Peter M. Schwarz, and Kathy Zeidenstein. SQL and Management of External Data. *SIGMOD Record*, 30(1):70–77, 2001.

[Moa97] Ryan Moats. RFC 2141: URN Syntax, 1997.

[Mv04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. `http://www.w3.org/TR/2004/REC-owl-features-20040210/`, 2004. W3C Recommendation.

[NE01] Shamkant B. Navathe and Ramez A. Elmasri. *Fundamentals of Database Systems.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[Neu05] Eric Neumann. Finding the Critical Path: Applying the Semantic Web to Drug Discovery and Development. *Drug Discovery World*, 6:25–33, 2005.

[NWQ+02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *Proceedings of the Eleventh International World Wide Web Conference, WWW2002, Honolulu, Hawaii, USA, 7-11 May 2002.*, pages 604–615. ACM Press, 2002.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[PC03] Cristian Pérez de Laborda and Stefan Conrad. A Semantic Web based Identification Mechanism for Databases. In *Proceedings of the 10th International Workshop on Knowledge Representation meets Databases (KRDB 2003), Hamburg, Germany, September 15-16, 2003*, volume 79 of *CEUR Workshop Proceedings*, pages 123–130. Technical University of Aachen (RWTH), 2003.

[PC05a] Cristian Pérez de Laborda and Stefan Conrad. Querying Relational Databases with RDQL. In *Berliner XML Tage*, pages 161–172. Humboldt-Universität zu Berlin, Freie Universität Berlin, 2005.

[PC05b] Cristian Pérez de Laborda and Stefan Conrad. Relational.OWL - A Data and Schema Representation Format Based on OWL. In *Conceptual Modelling 2005, Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005), Newcastle, NSW, Australia, January/February 2005*, volume 43 of *CRPIT*, pages 89–96. Australian Computer Society, 2005.

[PC05c] Christopher Popfinger and Stefan Conrad. Maintaining Global Integrity in Federated Relational Databases Using Interactive Component Systems. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part I*, volume 3760 of *Lecture Notes in Computer Science*, pages 539–556. Springer Verlag, 2005.

[PC06a] Cristian Pérez de Laborda and Stefan Conrad. Bringing Relational Data into the Semantic Web using SPARQL and Relational.OWL. In *Semantic Web and Databases, Third International Workshop, SWDB 2006, Proceedings of the 22nd International Conference on Data Engineering Workshops, ICDE 2006, 3-7 April 2006, Atlanta, GA, USA*. IEEE Computer Society, 2006.

[PC06b] Cristian Pérez de Laborda and Stefan Conrad. Database to Semantic Web Mapping using RDF Query Languages. In *Conceptual Modeling - ER 2006, 25th International Conference on Conceptual Modeling, Tucson, Arizona*, Lecture Notes in Computer Science. Springer Verlag, 2006. to appear.

[Pep00] Steve Pepper. The TAO of Topic Maps - finding the way in the age of infoglut. In *XML Europe - Proceedings of the XML Europe Conference, June, 12-16, 2000, Paris, France*, 2000.

[PH03] Zhengxiang Pan and Jeff Heflin. DLDB: Extending Relational Databases to Support Semantic Web Queries. In *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, volume 89 of *CEUR Workshop Proceedings*, 2003.

[PP04] Cristian Pérez de Laborda and Christopher Popfinger. A Flexible Architecture for a Push-based P2P Database. In *Tagungsband zum 16. GI-Workshop Grundlagen von Datenbanken, Mohnheim, NRW, 1.-4. Juni 2004*, pages 93–97. Universität Düsseldorf, 2004.

[PPC04a] Cristian Pérez de Laborda, Christopher Popfinger, and Stefan Conrad. DÍGAME: A Vision of an Active Multidatabase with Push-based Schema and Data Propagation. In *Enterprise Application Integration 2004, Proceedings of the GI-/GMDS Workshop on Enterprise Application Integration (EAI-04), Oldenburg, Germany, February 12-13, 2004*, volume 93 of *CEUR Workshop Proceedings*, 2004.

[PPC04b] Christopher Popfinger, Cristian Pérez de Laborda, and Stefan Conrad. DÍGAME: A Push-based P2P Database. In *Key Technologies for Data Management, 21st British National Conference on Databases, BNCOD 21, Edinburgh, UK, July 7-9, 2004, Proceedings Volume 2*. Heriot Watt University, 2004.

[PPC04c] Christopher Popfinger, Cristian Pérez de Laborda, and Stefan Conrad. Link Patterns for Modeling Information Grids and P2P Networks. In Il-Yeol Song and Stephen W. Liddle and Tok Wang Ling and Peter Scheuermann, editor, *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 8-12, 2004 Proceedings*, volume 3288 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[PPC05] Cristian Pérez de Laborda, Christopher Popfinger, and Stefan Conrad. Dynamic Intra- and Inter-Enterprise Collaboration Using an

Enhanced Multidatabase Architecture. In *16th International Workshop on Database and Expert Systems Applications (DEXA 2005), 22-26 August 2005, Copenhagen, Denmark*, pages 626–631. IEEE Computer Society, 2005.

[PR04] Johan Petrini and Tore Risch. Processing Queries over RDF views of Wrapped Relational Databases. In *1st International Workshop on Wrapper Techniques for Legacy Systems, WRAP 2004, Delft, Holland*, 2004.

[PS05] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/2005/WD-rdf-sparql-query-20051123/`, 2005. W3C Working Draft.

[PS06a] Eric Prud'hommeaux and Andy Seaborne. BRQL - A Query Language for RDF. `http://www.w3.org/2004/07/08-BRQL/`, 2006.

[PS06b] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. `http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/`, 2006. W3C Working Draft.

[PSH04] Peter F. Patel-Schneider and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax Section 2. Abstract Syntax. `http://www.w3.org/TR/2004/REC-owl-semantics-20040210/syntax.html`, 2004. W3C Recommendation.

[PZC06] Cristian Pérez de Laborda, Matthäus Zloch, and Stefan Conrad. RDQuery - Querying Relational Databases on-the-fly with RDF-QL. In *Posters and Demos of the 15th International Conference on Knowledge Engineering and Knowledge Management, EKAW 2006, Podebrady, Czech Republic*, 2006. to appear.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[RCJ+92] Ramana Rao, Stuart K. Card, Herbert D. Jellinek, Jock D. Mackinlay, and George G. Robertson. The Information Grid: A Framework for Information Retrieval and Retrieval-Centered Applications. In *Proceedings of the 5th Annual Symposium on User Interface Software and Technology (UIST'92)*, pages 23–32, Monterey, CA, 1992.

[SD02] Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In Ian Horrocks and James A. Hendler, editors, *The Semantic Web - ISWC*

2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings, volume 2342 of *Lecture Notes in Computer Science*, pages 364–378. Springer Verlag, 2002.

[SDWW01] A. Th. (Guus) Schreiber, Barbara Dubbeldam, Jan Wielemaker, and Bob Wielinga. Ontology-Based Photo Annotation. *IEEE Intelligent Systems*, 16(3):66–74, 2001.

[Sea04] Andy Seaborne. RDQL - A Query Language for RDF. `http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/`, 2004. W3C Member Submission.

[SKS98] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[SL03] Kai-Uwe Sattler and Frank Leymann. Information Integration und Semantic Web. *Datenbank-Spektrum*, 3(6):5–6, 2003.

[SM96] Heather A. Smith and James D. McKeen. Object-Oriented Technology: Getting Beyond the Hype. *DATA BASE*, 27(2):20–29, 1996.

[SMQ06] Susie Stephens, Alfredo Morales, and Matthew Quinlan. Applying Semantic Web Technologies to Drug Safety Determination. *IEEE Intelligent Systems*, 21(1):82–86, 2006.

[SPZ98] Stefano Spaccapietra, Christine Parent, and Esteban Zimányi. Modeling time from a conceptual perspective. In *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, USA, 1998*, pages 432–440. ACM, 1998.

[SS06] Steffen Staab and Heiner Stuckenschmidt, editors. *Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information*. Springer Verlag, 2006.

[SSC05] Gunter Saake, Kai-Uwe Sattler, and Stefan Conrad. Rule-based schema matching for ontology-based mediators. *Journal of Applied Logic*, 3(1):253–270, 2005.

[Sto01] Heinz Stockinger. Distributed Database Management Systems and the Data Grid. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, Washington, DC, 2001.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[SW00] Harald Schöning and Jürgen Wäsch. Tamino - An Internet Database System. In Carlo Zaniolo, Peter C. Lockemann, Marc H. Scholl, and Torsten Grust, editors, *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 383–387. Springer Verlag, 2000.

[TB93] Wouter B. Teeuw and Henk M. Blanken. Control versus Data Flow in Parallel Database Machines. *IEEE Transactions on Parallel and Distributed Systems*, (4):1265–1279, 1993.

[TC97] Can Türker and Stefan Conrad. Towards Maintaining Integrity of Federated Databases. In *Data Management Systems, Proceedings of the 3rd Int. Workshop on Information Technology, BIWIT'97, July 2–4, 1997, Biarritz, France*, pages 93–100, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[TF76] Robert W. Taylor and Randall L. Frank. CODASYL Data-Base Management Systems. *ACM Computing Surveys (CSUR)*, 8(1):67–103, 1976.

[The04] The Apache DB Project. Torque. `http://db.apache.org/torque/`, 2004.

[Tür03] Can Türker. *SQL:1999 & SQL:2003 - Objektrelationales SQL, SQLJ & SQL/XML*. dpunkt, 2003.

[Ull97] Jeffrey D. Ullman. Information Integration Using Logical Views. In *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer Verlag, 1997.

[Upd05] Andrew Updegrove. The Semantic Web: An Interview with Tim Berners-Lee. Consortium Standards Bulletin, June 2005. `http://www.consortiuminfo.org/bulletins/semanticweb.php`.

[VJBCS97] Pepjijn R. S. Visser, Dean M. Jones, Trevor J. M. Bench-Capon, and Michael J. R. Shave. An analysis of ontological mismatches: Heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering, Stanford, USA*, 1997.

[Wal03] Norman Walsh. RDF Twig: Accessing RDF Graphs in XSLT. In *Proceedings of the Extreme Markup Languages 2003 Conference, 4-8 August 2003, Montréal, Quebec, Canada*, 2003.

[Wik06] System One - Wikipedia3. `http://labs.systemone.at/wikipedia3`, June 2006.

[Win78] Elizabeth Winey. Data Flow Architecture. In *Proceedings of the 16th annual Southeast regional conference, Atlanta, Georgia April 13 - 15, 1978.*, pages 103–108. ACM Press, 1978.

[WM02] Richard Widhalm and Thomas Mück. *Topic Maps. Semantische Suche im Internet.* Xpert.press. Springer Verlag, Berlin, 2002.

[WPS+00] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database Replication Techniques: a three parameter classification. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), 16-18 October 2000, Nürnberg, Germany*, pages 206–215. IEEE Computer Society, 2000.

[XE04] Li Xu and David W. Embley. Combining the Best of Global-as-View and Local-as-View for Data Integration. In *Information Systems Technology and its Applications, 3rd International Conference ISTA'2004, June 15-17, 2004, Salt Lake City, Utah, USA, Proceedings*, volume 48 of *LNI*, pages 123–136. GI, 2004.

[XSL99] XSL Transformations (XSLT). `http://www.w3.org/TR/1999/REC-xslt-19991116`, 1999. W3C Recommendation.

[YPK03] Jian Yang, Mike P. Papazoglou, and Bernd J. Krämer. A Publish/Subscribe Scheme for Peer-to-Peer Database Networks. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 244–262. Springer Verlag, 2003.

[Zlo05] Matthäus Zloch. Automatic translation of RDQL queries into SQL. Bachelor's Thesis, Heinrich-Heine-Universität Düsseldorf, 2005.

List of Figures

2.1 A relation with four attributes and three rows 9
2.2 Result of the $\sigma_{City='Berlin'}(Address)$ operation 10
2.3 A sample RDF graph 19
2.4 A sample RDFS graph 20
2.5 Triple representation of the sample RDF graph 20
2.6 RDF/XML representation of the sample RDF graph 21

3.1 Three layers of abstraction using Relational.OWL 29
3.2 Main concepts of the Relational.OWL ontology 36
3.3 Schema representation using the Relational.OWL ontology 38
3.4 Data representation using a tailored schema ontology 40
3.5 XML data generated with `Rec2XML` 41
3.6 Amount of data after a data export 42
3.7 Mapping process 59
3.8 Sample mapping query 61
3.9 Sample mapping result 62

4.1 Multi-Peer-to-Multi-Peer data exchange 68
4.2 The `db` URI scheme 72
4.3 XML data exchange document 73
4.4 XML data exchange document with our novel URI 74
4.5 Relational.OWL-based schema file using our novel URI 76

5.1 The Export tab of the Relational.OWL application 81
5.2 RDQuery system architecture 87
5.3 Sample query translation using the GUI 89
5.4 Collaborative work realized with DÍGAME 92
5.5 DÍGAME architecture 95
5.6 Design of the DÍGAME wrapper 100

6.1 DLML components 108
6.2 DLML example 109
6.3 Link Pattern Catalog Classification 111
6.4 Elementary Link Patterns 114

6.5 Data Independent Link Patterns 115
6.6 Data Sensitive Link Patterns 117
6.7 Example using Link Patterns 119

List of Tables

2.1 RDQL clauses 24
2.2 Main SPARQL clauses 25

3.1 Classes defined in the Relational.OWL ontology 37
3.2 Properties defined in the Relational.OWL ontology 37
3.3 Average execution time for a SPARQL mapping 63