A Model-driven Approach for Building Distributed Ontology-based Web Applications

Richard Vdovják
A Model-driven Approach for Building Distributed Ontology-based Web Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen
op maandag 20 juni 2005 om 14.00 uur

door

Richard Vdovják

geboren te Liptovský Mikuláš, Slowakije
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. P.M.E. De Bra
en
prof.dr.ir. G.J.P.M. Houben

Copromotor:
prof.dr. J. Paredaens
Acknowledgements

First, I would like to thank my supervisors prof. Paul De Bra and prof. Geert-Jan Houben. I am thankful to Paul De Bra for providing his helpful guidance along the entire course of my PhD and I will also pleasantly remember long discussions concerning our shared hobby, photography. I owe many thanks to Geert-Jan Houben for his valuable input and for the pleasant working environment he created for me. He was always just a door away when I wanted to verify my research ideas or when I needed advice concerning life in general. Further, I would like to thank my co-supervisor prof. Jan Paredaens who provided useful comments on many database-related issues, especially those concerning the development of the RDF algebra.

I also want to thank all the people I collaborated with during my PhD research. I am grateful to my colleagues Flavius Frasincar and Peter Barna for their contribution to the many joint papers concerning the Hera system. I would like to thank Heiner Stuckenschmidt, Jeen Broekstra, and Ad Aerts for the numerous discussions regarding distributed RDF querying, which resulted in a number of joint publications, and also for their support with the distributed tests we conducted. Many thanks to my master student Pieter Bellekens for his implementation effort on our distributed mediator prototype, and to Kees van der Sluijs for his help with our distributed tests and for the translation of the summary of this thesis. I would also like to thank Reinier Post and Pim Lemmens for solving my computer-support related issues.

A special word of thanks goes to Anca Bucur, not only for proof-reading my manuscript and giving many valuable comments, but also for her understanding and encouragements during the years I had been conducting my PhD research.

Finally, I would like to warmly thank my parents and my sister for their continuous support. Their belief in my success was often the source of energy I needed to complete this dissertation. Regretfully, for my father these words of gratitude come too late.
Contents

1 Introduction ........................................... 1
   1.1 Motivation .......................................... 1
   1.2 Research Questions .................................. 2
   1.3 Outline of the Dissertation .......................... 4

2 Hera Web Engineering Framework ....................... 7
   2.1 Web Information Systems ............................ 7
      2.1.1 The Semantic Web Initiative .................. 9
   2.2 The Hera Framework ................................ 11
      2.2.1 Hera Layers .................................. 11
   2.3 Hera Methodology ................................... 13
      2.3.1 Conceptual Design .............................. 13
      2.3.2 Application Design .............................. 17
      2.3.3 Presentation Design ......................... 20
      2.3.4 Creating Custom Layouts ...................... 24
   2.4 Hera Software Suite ................................ 24
   2.5 Related Work ....................................... 29
      2.5.1 RMM - Relationship Management Methodology .... 29
      2.5.2 OOHDM - Object-Oriented Hypermedia Design Methodology 30
   2.6 Hera Summary ...................................... 32

3 Hera Integration Framework ....................... 35
   3.1 The Hera Integration Context ...................... 35
      3.1.1 Hera Requirements for Integration ............ 36
   3.2 Hera Integration Framework ....................... 38
   3.3 Integration Design .................................. 40
   3.4 Related Work ....................................... 50
   3.5 Integration Framework Summary ..................... 53

4 Distributed RDF Query Processing .................. 55
   4.1 Integration Architectures .......................... 55
      4.1.1 Basic Mediator Architecture .................. 56
      4.1.2 Hierarchical Mediator Architecture .......... 57
CONTENTS

4.1.3 Cooperative Mediator Architecture .................................. 60
4.2 Query Processing ......................................................... 63
  4.2.1 Data Structures .................................................... 64
  4.2.2 Source Index Hierarchies ......................................... 65
  4.2.3 Basic Query Answering Algorithm ................................ 68
  4.2.4 From a Single Path to Ad-hoc Queries ............................ 69
4.3 Related Work ............................................................ 71
  4.3.1 Distributed Architectures ......................................... 71
  4.3.2 Index Structures for Object Models ............................... 72
4.4 Summary of Distributed RDF Query Processing ....................... 73

5 Relational Optimization Techniques for RDF Querying ............... 75
  5.1 Query Optimization ................................................... 76
    5.1.1 Relational Approach for Processing RDF Path Queries ........ 78
    5.1.2 Solution Space .................................................. 78
    5.1.3 Cost Model ....................................................... 80
    5.1.4 Heuristics for join ordering .................................... 82
    5.1.5 Making the Query Plan Robust .................................. 84
    5.1.6 From Chain Queries to General Join Queries .................... 85
  5.2 Hera Mediator Implementation ....................................... 89
    5.2.1 The Merger of the Hera Mediator and Sesame ................... 91
    5.2.2 Performance Evaluation ......................................... 92
  5.3 Related Work .......................................................... 95
  5.4 Summary of Optimization Techniques ................................. 96

6 RAL: a Graph-based RDF ALgebra ........................................ 97
  6.1 RDF Data model ....................................................... 97
    6.1.1 RDF model ....................................................... 98
    6.1.2 Nodes and edges ................................................ 99
    6.1.3 RDFS ........................................................... 100
    6.1.4 Class and Property nodes ..................................... 102
    6.1.5 Complete models ............................................... 103
  6.2 Basic RAL operators .................................................. 103
    6.2.1 Extraction operators ........................................... 106
    6.2.2 Loop operators ................................................. 108
    6.2.3 Construction operators ........................................ 109
  6.3 Additional RAL features .............................................. 111
    6.3.1 Variables ....................................................... 111
    6.3.2 Additional Operators .......................................... 113
    6.3.3 RQL and RAL ................................................... 113
  6.4 RAL equivalence laws ................................................ 114
  6.5 Related work .......................................................... 117
  6.6 RAL Summary .......................................................... 119
Chapter 1

Introduction

This dissertation addresses several research problems associated with providing support for Web information system design and engineering, and the integration aspects within this process in particular. This chapter presents a brief outline of the motivation and research objectives for this dissertation.

Section 1.1 provides the motivation and a short overview of our research. In Section 1.2 we present the research questions that this dissertation addresses. In Section 1.3 we explain the structure of the thesis and relate the remaining chapters to our research questions.

1.1 Motivation

The overwhelming success of the World Wide Web has caused a change in the notion of information system by adopting the Web paradigm both as the delivery platform and as the source of its data. The Semantic Web initiative opens new possibilities which the "old" World Wide Web could not deliver. It also brings, however, a new set of requirements for information system design.

Designing large-scale Web information systems which are to be deployed on the Semantic Web platform, requires the use of suitable design methodologies. Such methodologies often advocate a model-driven approach, inspired by the separation-of-concerns principle. In order to tackle the complexity of the problem, each model in the system targets a different aspect of the design and often also a different level of abstraction. The underlying framework takes care of the transformations from instances of one model into instances of the next model in the pipeline, ultimately reaching the final result in the form of a presentation of the selected data elements.

We propose Hera, a model-driven methodology supporting Web information system design, focusing on the processes of data collection and presentation generation. All Hera models are based on RDF(S) [17; 74], the pivot language of the Semantic Web, making the methodology a suitable candidate for designing true Semantic Web information systems. When the content of such a system is to be gathered from different information sources, the specification of how data is to be retrieved and integrated requires an appropriate
specification framework and a suite of tools which are able to process this integration specification and to retrieve the data as a response to the user query.

The contribution of this thesis lies in designing the general Hera framework and in particular, in providing a solution to the problem of how to specify the integration of heterogeneous information sources on the Semantic Web in order to facilitate a uniform access to their distributed data. This carefully designed view of the selected content coming from different sources constitutes the semantic layer of the Hera framework, which is subsequently used by other modules to deliver a hypermedia presentation tailored to the end-user needs.

To be able to specify this semantic layer we propose an Integration Model formalism which is able to express a variety of semantic heterogeneities frequently occurring among sources on the Semantic Web. We designed and implemented a prototype of the integration engine which serves as a back-end of the Hera suite providing the semantic layer for the rest of the framework. Due to the fact that the proposed Hera architecture is modular with clearly defined interfaces built on top of the RDF foundation, the use of the integration engine is not limited solely to the Hera suite. It can also serve as a stand-alone general-purpose distributed RDF query engine allowing other parties to query a number of heterogeneous sources from a single semantically unified view, thus potentially extending the availability of the content on the Semantic Web.

As the integration engine often needs to process large amounts of RDF data, performance is an important issue. Several optimization techniques were adopted in the context of RDF in order to improve the speed of query processing: For intelligent query routing we designed specially tailored index structures which allow to query only the relevant (parts of) sources. For efficient result assembly, we applied join ordering techniques in order to optimize the process of combining chunks of information coming from different sources into a coherent result. We also proposed a special purpose RDF algebra (RAL) which is able to reason about RDF queries in terms of equivalent query plans and allows for algebraic optimizations in order to improve the efficiency of the query evaluation process.

1.2 Research Questions

In this section we identify several research questions that we address throughout the dissertation. Answering these questions effectively contributes to the state of the art in the associated areas. In particular, we contribute to the state of the art of Web engineering, integration of RDF data, distributed RDF query processing and optimization, and user interfaces for RDF. This dissertation provides answers to the following research questions:

**Research Question 1**

How can we support the design and implementation of Web information systems on the Semantic Web?

Considering the evolution of Web applications, we look at what is needed to be able
1.2. RESEARCH QUESTIONS

to engineer data-intensive Web information systems. The complexity of such systems makes it necessary to consider their design in smaller, clearly defined steps. Following the separation-of-concerns principle we want to identify the individual design phases and provide the description of their models. We also study what software components are needed in a suite that interprets such design specifications to be able to make an automated transition from the design models to a functional Web information system.

Research Question 2

How can we specify the integration of semantically heterogeneous RDF sources?

When a Web information system gathers its data from several semantically heterogeneous sources, integration becomes an important issue of WIS design. We first study the requirements introduced with the context of Web information systems and identify common semantic heterogeneities that frequently occur on the Semantic Web. Subsequently we investigate what integration primitives are needed in order to overcome these heterogeneities.

Research Question 3

Given the RDF integration specification, how can it be evaluated and what are the suitable architectures for that purpose?

After we identified our integration framework we need to consider what are the suitable ways of processing such specifications in order to be able to handle queries that span across multiple sources. This involves researching different architectures that can be used for the mediating purposes as well as designing suitable data structures for intelligent query routing.

Research Question 4

How can we optimize distributed RDF query processing?

The need for optimization comes with the large amounts of data that the integration engine needs to process. While proper index structures help to route queries to the right sources, we also need to investigate what is needed to improve the processing of results that come back from the sources. These results need to be often joined by the integration engine and we will investigate how this process can be optimized. Many times queries can be rewritten in a different yet equivalent form that yields the same results. We investigate a suitable algebraic formalism that would allow to reason about high-level RDF queries in terms of their query plans, facilitating their rewriting and further optimization.

Research Question 5

How can we help the user with exploration of the application content structure and subsequent query composition?

While the Semantic Web brings more structure into the Web content, it is not always
easy for the user to immediately grasp this structure, and even less so to be able to formulate queries over it. We need to look at peculiarities of RDFS and identify elements of a user interface that would be expressive enough to cover these peculiarities and at the same time would not overwhelm the user by its complexity. For that we compare suitable user interface metaphors and identify those that suit our purpose. We also investigate how can such interface assists the user with the formulation of his information need.

1.3 Outline of the Dissertation

To address the first research question, Chapter 2 introduces the Hera framework that includes both a model-driven methodology for Web information system design and an associated software suite which executes the design specifications in order to implement a concrete Web information system. Parts of the research presented in Chapter 2 were previously published in [40; 120; 39], co-authored by Flavius Frasincar, Geert-Jan Houben, and Peter Barna. Our specific contribution is the specification of the data collection phase and implementation of the semantic layer of Hera framework.

Chapter 3 addresses our second research question by detailing the Hera integration framework, proposing our integration model and its RDF(S) representation. This model is used by the integration engine to reconcile the semantic differences between semantically heterogeneous sources and the specified Web information system that uses them as its data providers. Chapter 3 reflects our previously published research from [121; 122], co-authored Geert-Jan Houben.

Chapter 4 answers the third research question; it discusses several architectures of distributed RDF query engines and details the adopted query processing. In this chapter we describe the mediator implementation; the Hera mediator is built on top of an existing RDF storage engine called Sesame [18] and acts like its distributed embodiment. The architectural discussion in Chapter 4 was inspired by [121], co-authored by Geert-Jan Houben. The parts on source index hierarchies were published in [111] and (an extended version) in [110], co-authored by Heiner Stuckenschmidt, Geert-Jan Houben, and Jeen Broekstra. Our specific contribution lies in extending the source index data structure to cover a number of arbitrary paths and facilitating the RDF query processing in the context of semantically heterogeneous sources.

Chapter 5 together with Chapter 6 provide an answer to the fourth research question. Chapter 5 approaches the problem of RDF query optimization from the relational point of view, treating an RDF graph as a number of connected relational tables which need to be joined. These joins, especially when the data is coming from different sources, need to be executed by the integration engine. In order to speed up this process, we deploy optimization heuristics that find an efficient order of the involved join operations. Parts of the research presented in this chapter appeared in [111; 110], co-authored by Heiner
1.3. **OUTLINE OF THE DISSERTATION**

Stuckenschmidt, Geert-Jan Houben, and Jeen Broekstra. Our specific contribution is in identifying the distributed aspects of RDF query processing, mapping RDF path queries to the a sequence of join operators and identifying our cost model, which enables the use of the join ordering optimizations. The initial approach only treated linear path expressions; we provide an extension to arbitrarily shaped path queries.

Chapter 6 treats RDF as a directed labeled graph and proposes an RDF (graph) algebra for reasoning about RDF queries and the way they are evaluated at higher level of abstraction. Note that these two approaches are complementary and can be combined in order to achieve a better performance. Chapter 6 is based on our research published in [41] and extended in [42], co-authored by Flavius Frasincar, Geert-Jan Houben, and Peter Barna; this chapter also appears in [36].

Chapter 7 addresses our last research question by proposing a user interface that helps the user to browse and get acquainted with an application’s conceptual model expressed in RDFS and at the same time it assists with the data request formulation. Parts of this chapter appeared in [119; 118].

Chapter 8 concludes the dissertation with a summary and suggestions for possible future extensions.
Chapter 2

Hera Web Engineering Framework

This chapter introduces the Hera framework consisting of a model driven methodology for engineering Web information systems and the associated software suite for executing the design models. Applying the separation-of-concerns principle, the Hera methodology distinguishes two principal design phases: the data collection phase and the presentation generation phase. These phases are further refined into several design steps, each addressing a different aspect of the Web information systems design process. Following the methodology, during every design step the designer creates a design model—an engineering artifact for that particular step. These models then serve as input for the Hera software suite which by interpreting them realizes the goal of the designer: the implementation of a Web information system.

The chapter is structured as follows. Section 2.1 establishes the notion of Web information system and describes the challenges that the designer meets during the process of creating such systems. Section 2.2 introduces the Hera framework, focusing on the Hera methodology. We explain the modeling foundations of Hera, detailing the individual steps of the conceptual, application, and presentation design. Section 2.4 provides an overview of the associated Hera software suite. Section 2.5 describes related work in the field of Web engineering, detailing some of the representative examples. We conclude this chapter with a brief summary and discussion in Section 2.6.

2.1 Web Information Systems

Considering the number of its users and the attention that it attracts, it is fair to say that the World Wide Web is one of the most popular information channels of today. It reaches a very diverse audience on different platforms worldwide and 24 hours per day. Its success is overwhelming and its impact on the humankind is tremendous. Some compare its importance with Guttenberg’s invention of the printing press. The World Wide Web shaped the landscape of today’s information society and we can already see how different areas such as science, business, education, human interaction, or entertainment are being
influenced by this (almost) ubiquitous information library and publishing/communication platform.

The early applications on the Web presented data in terms of carefully authored hyper(media) documents. Typically, the author hand-crafted a static collection of pages and links between these pages in order to convey information to the users. At the time, this use of hypermedia was already a step forward and contributed to the popularity of the Web. As more and more (existing) data sources were connected to the Web, more information became available and the typical Web application became data-intensive. A data-intensive Web application uses for example data generated from structured files or databases, usually in an on-the-fly manner, in order to deliver the right information to the user. In this context, the hypermedia presentation cannot be hand-crafted anymore; instead, the application has to generate a presentation in an ad-hoc fashion from the data that satisfies the user query.

This trend has significantly influenced and changed the notion of information system. Many existing information systems were ported onto the Web platform: a modern professional information system has become a data-intensive Web application. Throughout this dissertation we use the term Web Information Systems (WIS) coined by [58]: WIS are information systems that use the Web paradigm (and technologies) to retrieve information from the information sources and to deliver it to the users.

It is typical for WIS that they need to bridge the gap between a collection of heterogeneous and dynamic data sources, and a group of users with different preferences using different platforms for accessing the information. This aspect of WIS makes the early proposals for designing and implementing Web or hypermedia applications not applicable anymore [88]. Developing a hyperdocument typically meant a mix of content and presentation design, but also a lot of ad-hoc programming. The data-intensive and dynamic nature of WIS requires a more rigorous development process. One reason is that the one-size-fits-all approach that is characteristic for traditional hypermedia is not suitable for delivering information at run-time to different users with different platforms (e.g. PC, PDA, WAP phone, WebTV) and different network connections (e.g. dial-up modem, network copper cable, network fiber optic cable).

A characteristic aspect of WIS is the automatic generation of a hypermedia presentation of the data that the application delivers to the user. As mentioned above, due to the dynamic nature of the data the hypermedia output needs to be generated automatically by the application. A crucial issue in the application design is therefore the specification of this hypermedia generation process. True Web engineering approaches offer designers and programmers a design framework often based on a model-driven approach, specifying the different aspects of the complete application design in terms of separate models, e.g. for data (content) and hypermedia presentation.

Personalization is also typical for the hypermedia design in WIS, taking into account
the requirements from the users, e.g. user preferences, and the requirements from their platforms, e.g. device capabilities and network connection. This personalization is one example of adaptation [22] in the design process. Adaptation tries to overcome the problems of the one-size-fits-all approach in the traditional hypermedia. It does so by generating a hypermedia presentation that suits the browsing context and user preferences: we say that the presentation is adaptable. Adaptation can also be based on the user’s browsing history: in that case we say that the presentation is made adaptive.

Integration is another characteristic feature for WIS [122]. Usually, the content of the system is gathered from different sources that are distributed over the Web. This requires that the design of the WIS includes specifications of which sources to select and how to map the data in the sources to the data (model) in the WIS (both in terms of schema integration and data integration). With such specifications, the WIS can retrieve the data that needs to be included in the presentations that are generated. The approaches and techniques used for integration are inspired by similar approaches used for the traditional integration of databases; their application in the context of WIS proves to be useful for WIS design.

2.1.1 The Semantic Web Initiative

A modern Web information system is not only a self-standing data-intensive Web application. It often needs to interact with other WIS, either to gather data from them or to serve as data provider itself. This interaction inevitably requires to achieve at least some degree of interoperability among the different systems. The initial Web was however designed mainly with the human audience in mind, and HTML [94] was both the presentation and the content language. While suitable for humans, this approach proved to be one of the major bottlenecks for reaching interoperability among machines (software applications) that needed to process Web information. In order to address this bottleneck, the WWW is moving towards a new generation — the Semantic Web. According to Berners-Lee, the founder of the WWW and one of the main initiators of the Semantic Web initiative: “The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.” [8].

While XML [16] provides the syntactic foundation of the Semantic Web, it does not address the main issue of interoperability, i.e. the machine processable semantics. The first step towards addressing this issue is the (meta)data format called Resource Description Framework [74].

RDF and RDFS

The Resource Description Framework (RDF) is a language for representing information on the Web. “RDF is designed to represent information in a minimally constraining, flexible way. It can be used in isolated applications, where individually designed formats might be
more direct and easily understood, but RDF’s generality offers greater value from sharing. The value of information thus increases as it becomes accessible to more applications across the entire Internet.” [67]

RDF is centered around the notion of triplet, the information atom of RDF consisting of three parts with clearly defined roles: subject—the (Web) resource that is being described, predicate—the property of the subject, and object—the value of the property. By organizing (Web) information into triplets like these, RDF introduces a small albeit very extensible semantic commitment: the information is split into subjects, their properties, and the property values. The latter can in turn be subjects with their own properties and values. With the above semantic commitment, RDF essentially paves the foundation of the Semantic Web: new layers offering more semantics are built on top of RDF. One such layer is RDF Schema (RDFS) [17], a language for defining RDF vocabularies.

As the name suggests, RDF Schema is intended for defining application specific vocabularies or schemata. These vocabularies define taxonomies of resources and properties such that they subsequently can be used by specific RDF descriptions. RDFS is designed as a flexible language to support distributed description models. Unlike XML DTD or XML Schema [30; 114], RDFS does not impose a strict typing on descriptions: for example, one can use new properties that were not present in the schema, a resource can be an instance of more than one class, etc.1

RDFS is frequently used to express schemata that describe at intensional or schema level the content provided by different Web applications. These domain descriptions consisting of hierarchies of classes/concepts and their properties/relations are often referred to as (light-weight) ontologies. Indeed, when we adopt Gruber’s definition and consider an ontology as being “an explicit specification of a conceptualization” [46], RDFS can be seen as a (light-weight) ontology language. Of course, RDF(S)2 itself does not provide means to express everything that a knowledge engineer may deem necessary. It is clearly less expressive than some other ontology languages such as OWL [117]. On the other hand, its simplicity allows for both easier tool support and broader adoption by the developers community (especially at this stage of the Semantic Web evolution). Being a W3C recommendation, RDF(S) became de-facto a Web (meta)data standard.

We also embraced RDF(S) as the core (meta)data language in Hera—our framework for supporting WIS design and implementation. We argue that modern WIS can benefit greatly from the use of Semantic Web technologies like RDF and RDFS. The use of such technologies in the area of WIS and their subsequent deployment on the Semantic Web platform constitute a natural evolutionary step towards Semantic Web Information Systems (SWIS).

1RDF and RDFS is a reoccurring theme in this dissertation, and we provide here only a general description. We address RDF/RDFS in more detail in the chapters to come, in particular in Chapter 3 where we describe our RDF integration framework and in Chapter 6 where we describe our RDF Algebra.

2By RDF(S) we refer both to RDF and RDFS.
2.2 The Hera Framework

The main target of the Hera framework is to provide support for WIS design and implementation. A typical Web information system generates a hypermedia presentation from data that is gathered from several, possibly heterogeneous, data sources as a response to a user query. This entire process of retrieving data and presenting it in hypermedia format needs to be specified during the design phase and the underlying software suite has to automatically execute that specification during runtime. Taking this into account, the Hera framework includes both a design methodology and a suite of software tools that help to transform the design specification into a working Web information system.

Adopting the separation-of-concerns principle, the Hera approach distinguishes the content data from the navigational structure and from the presentational issues. The Hera methodology distinguishes two principal phases: data collection, and presentation generation. Each of the two phases consists of several different data transformations that are necessary in order to generate the hypermedia output in response to a user query. The methodology uses different models for the conceptual content and for the hypermedia aspects of the application, and makes integration and adaptation central issues in its models.

The associated Hera software suite includes a number of steps to get from data retrieval to presentation generation at the level of instances. These steps basically build a sequence of data transformations that in the end present to the user the right data in the right format (e.g. HTML [94], WML [35] or SMIL [3]). Considering the advantage of Web application interoperability, we have chosen to use Semantic Web technology, specifically RDF(S) [74; 17].

RDF(S) is the main format used in our data transformations. One of the reasons for choosing RDF(S) is that it is a flexible (supporting schema refinement and description enrichment) and extensible (allowing the definition of new resources/properties) framework that enables Web application interoperability. An example of application interoperability is the usage of different navigation models for a given application domain ontology. In Hera, model instances are represented in plain RDF and validated against their associated models (schemata) represented in RDFS.

The use of RDFS also allows us to reuse existing RDFS vocabularies like the Dublin core initiative [28], User Agent Profile (UAProf) [125], and Composite Capability/Preference Profiles (CC/PP) [68]. With RDF(S) we also have an effective format to deal with the semi-structured nature of the data involved. For the purpose of data retrieval we use one of the most advanced RDF(S) query languages to date, SeRQL [19; 50], and its java-based interpreter called Sesame [18]. This combination proves to be useful when building our retrieval engine, which in fact acts as a distributed SeRQL query engine.

2.2.1 Hera Layers

The typical structure of the WIS architecture in the Hera perspective is given in Figure 2.1. It includes three layers:
• The Semantic Layer defines the content that is managed in the WIS in terms of a conceptual model. This layer includes the definition of the process of integration needed to gather the data from different sources. If not restricted by the designer, the data from the semantic layer can be accessed by external software entities, such as a search agent or information retrieval engine.

• The Application Layer defines the abstract hypermedia (navigation) view on the data in terms of an application model, which represents the structure shown to the user in the hypermedia presentation. This layer includes the definition of the adaptation in the hypermedia generation, e.g. based on a user model and a user/platform profile. If desired, the Application Layer can interface not with the own Presentation Layer, but with an external Presentation Layer like the AMACONT presentation generation engine [33].

• The Presentation Layer defines the presentation details that together with the definitions from the Application Layer are needed for the generation of a presentation for a concrete presentation platform, e.g. HTML, WML or SMIL. It outputs this presentation to browsing platform of the user.
2.3 Hera Methodology

The Hera methodology prescribes a number of design steps to guide the designer through the WIS design process at the end of which he produces a WIS specification in the form of the models which are subsequently used by the Hera software suite to realize the newly designed Web information system. The Hera methodology distinguishes the following principal phases:

- data collection
- presentation generation

During the data collection phase the designer defines the semantic structure of the Web information system, captured in the form of a Conceptual Model (CM). The CM usually covers terms agreed within a certain domain used by a certain community for which the Web information system is intended. The CM consists of hierarchies of classes together with their properties. The conceptual model then needs to be connected to the available data sources by means of an Integration Model. These two models are the main engineering artifacts of the data collection phase.

During the presentation generation phase, the hypermedia aspects of the Web information system are captured in the form of the Application Model (AM). The Application model groups concepts from the CM into meaningful navigational units called slices, and creates a hypermedia structure over them taking into account the adaptation aspects of the WIS design. While the Hera suite can generate presentations directly from the AM, if the designer desires to specify also the look-and-feel of the presentation, he captures these presentation specifics in a Presentation Model (PM). The AM and PM are the main engineering artifacts of the presentation generation phase.

In the following sections we describe the main design steps and their engineering artifacts in more detail, leaving out the integration specifics, as this is the main topic of the next chapter.

2.3.1 Conceptual Design

During the conceptual design we capture the semantics of the domain that the intended WIS is going to describe. This design process is materialized in the form of a conceptual model. The CM is an ontology which explicitly captures the conceptualization of the domain of interest. In case the domain of interest is a complex field, such as the medical domain, the process of designing a CM often involves a collaboration between domain experts and knowledge engineers so that the resulting CM reflects accurately the knowledge from that particular field. To explicitly capture such knowledge already provides added value in the sense of a common semantic agreement, but the main advantage comes when the system fills the CM with instance data coming from different sources, which otherwise
would have to be individually accessed by the user. When combined with the integration apparatus, the conceptual model provides a uniform semantic view over multiple data sources that supply data to populate the CM. However, the CM itself is constructed independently of the source schemata, which may change. This independence has a positive consequence for the rest of the Hera framework: the presentation generation is shielded from the structural changes of the sources. In fact, the CM serves as an interface between the data collection phase and the presentation generation phase.

Figure 2.2: Conceptual model vocabulary.

Figure 2.3 represents the CM vocabulary which contains the following elements: concept, concept relationship, and concept attribute. Concepts capture the explicit conceptualization of a particular domain; they represent abstract entity types that typify groups of individuals—the data instances.

Concepts can have a number of concept relationships and concept attributes. The concept relationships represent properties that relate concepts to each other, while concept attributes are of certain Media types, such as Text, Image, or Audio. Following the (Web) standards such as XML Schema Datatypes [12] and MPEG-7 [81], these media types are further refined in a Hera media vocabulary, details of which are beyond the scope of this dissertation; we refer the interested reader to [37].
The running example used throughout this chapter describes the design of a Web Information System serving as a virtual art gallery that allows visitors to create on-the-fly exhibitions (browseable presentations) featuring their favorite painters, paintings, and painting techniques. These are assembled on demand, based on the visitor’s query, from the exhibits coming out of different (online) museums and annotated with relevant descriptions from an online art encyclopedia. All this data is offered from a single entry point, semantically represented by the CM. The Conceptual Model of our application defines (a part of) a virtual museum domain ontology and roughly corresponds to the actual museum catalog of the Rijksmuseum in Amsterdam\(^3\).

A part of this CM is presented in a graphical form in Figure 2.3. The CM consists of several concepts, relationships, and attributes.

Concepts are depicted as ovals; the more abstract concepts, such as Artifact and Creator are on the top part of the model. From those, more concrete concepts are inherited, e.g. Painting and Painter.

Relationships are denoted with full arrows and are modeled as RDF properties. Similarly to concepts, there are abstract relationships (properties) such as created_by and creates, and their more concrete subrelationships (subproperties) painted_by and paints.

Attributes, e.g. Creator.name: String, are depicted as rectangles and represent properties which have as their domain a concept, e.g. Creator, and as their range a basic media type, e.g. String.

\(^3\)http://www.rijksmuseum.nl
CHAPTER 2. HERA WEB ENGINEERING FRAMEWORK

Figure 2.4: Conceptual Model (RDFS)

Note that the graphical notation used in Figure 2.3 resembles the graph syntax of RDF(S), extended with the media types and with special edges representing built-in RDFS properties like `subClassOf` and `subPropertyOf`. Other (application) properties are denoted as full-arrow edges and are starting and ending directly in classes (denoted as ovals) that represent their domains and ranges respectively.

For the purpose of machine processing, the Hera framework uses an equivalent XML-RDF serialization syntax, an example of which is shown in Figure 2.4. During the actual run-time process of presentation generation the CM is on request populated with instances (RDF statements) which represent the query result that propagates to the presentation
2.3. HERA METHODOLOGY

generation phase. In Figure 2.5 we present a concrete example of instances adhering to the CM introduced in Figure 2.3. Note that the source data generally comes from different (heterogeneous) sources and that the actual instance retrieval is performed by the Integration Engine, details of which we present in Chapters 3 and 4.

```xml
<Painting rdf:ID="Painting_ID01">
  <name>
    <String>
      <data>The Stone Bridge</data>
    </String>
  </name>
  <year>
    <Integer>
      <data>1638</data>
    </Integer>
  </year>
  <picture>
      <data>On Canvas "The Stone Bridge"</data>
    </Image>
  </picture>
</Painting>
```

Figure 2.5: Conceptual Model Instance (RDF)

During the integration design the designer connects the conceptual model to the external sources so that their (differently structured) instance data can populate on request the CM. These instances propagate into the presentation generation where they are transformed into a hypermedia presentation suitable for the user platform and for the user preferences. The presentation generation is composed from two main steps: the application design producing a navigational view over the retrieved data and the presentation design addressing the look-and-feel of the presentation. In the following sections we focus on these two aspects of the WIS design.

2.3.2 Application Design

The WWW paradigm changed the way information systems present their data to the end-user. A “flat” report from database tables is no longer sufficient to convey the information to the user who wants to interact with the query result by navigating and browsing. Modern WIS typically provide a carefully designed navigational structure over the retrieved data. In Hera, the designer creates this navigational structure during the application design process, materializing it in the form of an Application Model (AM).
The AM describes a hypermedia view over the CM, i.e. it groups parts of the CM into presentation units and connects them with the navigational primitives from the application model vocabulary. The application model vocabulary presented in Figure 2.6, introduces the three main application modeling primitives: slice, slice relationship, and slice attribute. A slice is a meaningful presentation unit created by the designer in order to convey a certain presentation message. Hera distinguishes empty or constant slices, and non-empty or data-driven slices. The (instance) content of an empty slice is determined during the design and stays constant throughout the presentation. In case of a non-empty slice, the designer defines its content at schema level by associating it with the owning CM concept, the values of which are instantiated at runtime. Slices can be organized into complex slices creating a composition hierarchy at the top of which are the top-level slices that (may) serve as navigational targets and correspond to pages to be presented on the user’s display.

Figure 2.6: Application model vocabulary.

Slices can have a number of media attributes and slice relationships. Similarly to the CM, media attributes represent atomic values of different media types. The slice relation-
ships come in two flavors: *Slice Navigation* and *Slice Aggregation*. The slice navigation relationship represents the notion of a hyperlink connection from the anchoring slice to the target slice. The slice aggregation relationship indicates embedding of another slice. If two slices connected by an aggregation relationship have different owning concepts, the concept relationship from the CM on which the aggregation is based has to be specified. In case this concept relationship has multiple cardinality, the embedding of multiple target slices is realized via the *Set* construct which represents a collection of (instance) slices associated to one anchor slice. This can be refined later during the presentation design into a guided tour or an index presentation structure.

Figure 2.7 presents the AM for our example. It defines a slice navigation model composed of two complex slices and two slice navigation properties between them. The simple slices are depicted as ovals and the slice aggregation properties are shown using the nested composition notation. Since the relationship between the *Technique* concept and the *Painting* concept is one-to-many, we introduced a set of links (pictures of paintings) when navigating from *technique* to *painting*.

In order to realize adaptation the designer can associate appearance conditions to slice references [40]. The appearance conditions enable two kinds of AM adaptation: *conditional inclusion of fragments* (slices in our context) and *link hiding* [22]. A link is hidden when its destination slice has a condition that evaluates to *false*. The slice appearance conditions use attribute-value pairs from the user/platform profile.

The user/platform profile defines the device (display) capabilities and the user preferences. The user/platform profile is an RDF description that instantiates two CC/PP
vocabularies. As advocated in section 2.2, one of the important advantages of RDF(S) is the ability to reuse existing RDFS vocabularies. UAProf is such a vocabulary developed by WAP Forum to model device capabilities (e.g. `ImageCapable` attribute). A new CC/PP vocabulary was created to model user preferences (e.g. `levelOfExpertise` attribute).

Conditions that use the user/platform profile elements specify adaptability and conditions that use the user model elements specify adaptivity. Adaptability is done prior to the presentation browsing while adaptivity is done dynamically as the user model changes during the presentation browsing. The user/platform profile is static information (prior to presentation generation), while the user model represents dynamic information (generated on the fly as the user is browsing the presentation). Figure 2.8 gives an example of a condition for adaptability: the `description` attribute of the technique is removed from from the main slice if the user is not an expert.

![Figure 2.8: Adaptation (adaptability/adaptivity) in the application model.](image)

2.3.3 Presentation Design

During the application design, the designer specifies the hypermedia structure of the presentation from which the Hera software suite generates a (default) browseable presentation. However, besides navigation there are also other aspects that are important to create visually appealing presentations. In particular, the designer may desire to detail the “look-and-feel” of the presentation. These aspects are captured in the course of the presentation design and they are materialized in the form of a Presentation Model (PM).

The presentation model can be seen as a (presentation) view over the application model; the PM groups parts of the AM and associates some presentation properties, such as position of individual presentation elements, or in case of multimedia their timing and synchronization. The vocabulary of the presentation model is described in Figure 2.9. The main presentation primitives are `region`, `region attribute`, and `region relationship`. 
A region is defined as a collection of attributes and possibly other regions. This recursive definition facilitates nested regions, thus providing a means to reuse regions in a component-like manner. Every region is associated with an area where it should be displayed. An area is of a rectangular shape having as properties its dimensions and location.

A region can contain a number of attributes, properties that relate concrete (media) values to the region. There are two types of attributes: slice attributes and (constant) region attributes. A slice attribute acquires its value from the AM to which it is mapped and it changes its value throughout the presentation. A region attribute has its value determined at the design phase and stays intact throughout the presentation.

Figure 2.9: Presentation model vocabulary.

The region relationships materialize (more abstract) slice relationships into the following types: navigational, temporal, and spatial. Graphically, region relationships are depicted as arcs with arrows, where the style of the arc indicates the type of the relationship as depicted in Figure 2.10.
Navigational Relationships

The navigational relationships represent the classical hyperlinks (so called 'click-able' links). When the user clicks on the source region the target region is invoked and displayed, and the source region either vanishes or stays, depending on its type.

It is also possible to guard the execution of the link with a condition (the link is followed only if the condition is satisfied) or to specify an event when the link should be followed. The default event for navigational links is the mouse-click event, but the designer can choose different events (e.g. mouse-over). Navigational links are graphically depicted with solid arcs.

Another refinement of the presentation model is that both region attributes and relationships can be either source-vanishing or source-preserving. The first case represents a classical link-following behavior, where after traversing the link the anchor/source region is replaced with the destination region—the source vanishes. In the second case, when a source preserving attribute or relationship is invoked (traversed), the source region is kept, i.e. the anchor of the link is still present. This behavior is often useful when the designer wants to show both a general and (on request) a detailed description of a certain presentation element. For instance, Figure 2.11 shows the main region presenting the technique attributes with a list of artifact names that exemplify the particular technique. The name attribute serves as anchor to the Artifact region which is shown (in addition to the already shown list of names) when the user invokes this link. Graphically, an arc starting with a solid circle indicates a persistent (not vanishing) source region.
2.3. HERA METHODOLOGY

Temporal Relationships

Temporal relationships were introduced in order to express the notion of time in the specification of multimedia presentations. Temporal links can be used to describe both intrinsic delay (a duration of a video or audio clip [52]) and presentation time (temporal links introduced by the designer). There is a time-out event associated with every temporal relationship. After the time elapses, the relationship is realized, i.e. the destination region is invoked) \(^4\). Temporal links are depicted with dashed arcs and can be organized in parallel or sequential compositions.

A sequential composition is depicted as a chain of links with one starting node (starting source region), where the destination region of one link is the source region for the next link. Two temporal relationships are considered to be in parallel composition if they are not included in the same sequential composition. Figure 2.12 shows an example of three sequential chains (A\(^\ast\), B\(^\ast\), C\(^\ast\)), which are organized into a parallel composition and synchronized before they enter the region D. Note that normally the starting region from a sequence is shown at once (A1, B1) but it is also possible to specify a delay with respect to the top region (C1).

Except of the synchronization of the temporal relationships, our framework offers also multidimensional relationship synchronization where relationships of different kinds are synchronized. This gives the designer the opportunity to specify new types of dependencies among relationships. For example, in Figure 2.13 a navigational link from the region ‘Text’ is synchronized with a temporal relationship coming from the region ‘Audio’. The target region is invoked only after both events from the involved relationships (mouse-click, time-out) occur.

Spatial Relationships

Spatial relationships implement the layout characteristics of a presentation. They also facilitate the description of more complex presentation elements like indices, guided tours, etc. A spatial relationship describes a relative position of one region with respect to another region\(^5\). The semantics of spatial links is slightly different compared to the one of

\(^4\)The synchronization can sometimes introduce an additional delay (the time-out of one ‘track’ elapses but it still has to wait for the other with which it is synchronized).

\(^5\)The point of reference being the coordinates of the left upper corner of the target region
navigational and temporal links. The notion of link following is missing here, e.g. both the source and the target of a spatial relationship are displayed at once, provided the condition and the event associated with the link permit so. Spatial links are graphically depicted with dotted arcs.

### 2.3.4 Creating Custom Layouts

A region which contains a set slice relationship is instantiated during the presentation to several (sub)regions with containing the collection elements.

In order to be able to specify at type level how these instances are presented we introduce so-called self-relationships. The idea is that when a (collection) region contains a relationship to itself, this link applies to the collection elements within that region. Since there might be more of such relationships needed to describe the desired presentation structure, there must be a unique order in which they are applied. To facilitate this, the ordering of relationships was introduced. In Figure 2.14 we provide two examples to illustrate this approach. The presentation models are depicted on the left and the respective screen renderings on the right.

Figure 2.14 (bottom) describes a bookcase region. It consists of the shelf region, which contains the thumb region that has a navigational link to the picture region and also a link to itself saying that the next thumb will be placed to its right. This rule is applied until there is no room left in the shelf (the horizontal dimension of the 'shelf' region). When the shelf is full we proceed with the next one by applying the self spatial relationship of the 'shelf' region which has the lowest priority (2).

### 2.4 Hera Software Suite

The models presented above constitute the input for the Hera software suite. This section provides an overview of the particular Hera software modules that are involved in the data transformations necessary to convert “raw” source data into a full-featured hypermedia presentation.

When combined, the different Hera models create a layered structure where the above layer uses the layer below, e.g. the application model is built on top of the conceptual model. This has a significant advantage: it facilitates model-driven transformations. The Hera software suite is closely related to the Hera methodology and implements these different
data transformations that are necessary to generate the hypermedia output in response to a user query.

The first phase—data collection, helps to make available the data from different sources, such that in response to a user query a conceptual model instance is generated that contains the data for which the application is going to generate a presentation: see Figure 2.15.

The integration is in principle performed before querying, as opposed to the retrieval and presentation generation that are performed for every query. The integration represents the stored data, and therefore uses an integration model to map the data from the different sources into concepts of the conceptual model. From a mapping at schema (ontology) level a mapping at instance level is derived. This mapping is needed whenever for a given query the instances that compose the query result need to be retrieved. These instances need to be extracted by the mediator from the different source ontology instances. The role of the integration is to make (on-demand) these source ontology instances available.

---

Figure 2.14: Defining a bookcase layout.

---

6In these figures the ellipses denote the transformations (in XSLT or Java) and the squares denote models or data. The shapes in grey denote application-independent items, the shapes in white with bold lines denote application-dependent items, while the others are query-dependent items.
The data retrieval handles the reception of the user query and in response produces a conceptual model instance for the query result. It starts with the translation of the query formulated by the user into a query that can act as a retrieval request on the stored data. This translation takes into account that while the user is allowed to formulate a query by mentioning items from the conceptual model or application model, the application model defines exactly which concepts need to be retrieved in connection with the items mentioned: this is known as query extension. Subsequently, using the query engine, the mediator retrieves the data from the sources and provides the query result. Finally, this query result needs to be transformed into the conceptual model instance that is passed on to the phase of presentation generation.

In the presentation generation phase, a hypermedia presentation is generated (see Figure 2.16) for the retrieved data. It starts from the data that constitutes the result of the user query, represented by the conceptual model instance. This data is transformed in three main steps into a presentation suited for the user’s platform.

First step is the generation of the so-called cmi2ami transformation responsible for translating the CM instances into instances of the AM. This transformation is derived from the application model and the user/platform profile considering adaptation. In the second
step this newly built transformation is used to transform the instances of the conceptual model into the application model instances that adhere to the application model. During the last step, a presentation for a specific platform is generated and applied; the adapted application model instance is serialized in the specific format of the user’s platform (e.g. HTML, WML, or SMIL). Figure 2.17 gives three snapshots of the hypermedia presentations generated for a HTML, WML, and SMIL browser. For each type of serialization a specific stylesheet is used.

In the code generation we used a media-directed translation scheme: for each media type appropriate code is generated. For example, strings were represented in normal font and integers in italic font. For the WML browser, images are removed and one may need to scroll down in order to view the full text. A back button similar to the back button from existing HTML/SMIL browsers was implemented for the WML serialization.
When the AM template needs to be adapted based on the specified slice appearance conditions, this substep of the AM adaptability is executed by an appropriate stylesheet. This stylesheet has two inputs: the AM template and the user/platform profile. The user/platform attributes are replaced in the conditions by their corresponding values. The slices whose condition evaluate to \textit{false} are discarded, and the hyperlinks pointing to these slices are disabled. For the example depicted in Figure 2.8 the \textit{description} attribute will be suppressed for users who are not experts. In Figure 2.16 this substep is labeled 2.2.

![Figure 2.17: Hypermedia presentation in different browsers.](image)

The AM adaptability conditions use information from the device capabilities and user preferences stored in the user/platform profile. One can also generate adaptive hypermedia presentations by considering a user model that represents the user’s state of mind based for example on the user’s history of presentation browsing.

Hera adaptive features are designed in the spirit of AHAM (Adaptive Hypermedia Application Model) [15], a Dexter-based reference model for adaptive hypermedia. AHAM defines in the Storage Layer three models: the domain model, the user model, and the
2.5. RELATED WORK

adaptation model. In Hera the above models have two parts: a model-based part and an instance-based part. The model-based part describes the model using schema elements (coming from CM/AM) that will be later on instantiated with data (coming from CM/AM instances). In order to generate adaptive hypermedia presentations the Hera framework can take advantage of the AHA! [14]—a general purpose adaptation engine. For further details on Hera-AHA! binding we refer to [120].

In our approach we primarily focused on hypermedia applications. This implies that the main interaction from the user is based on navigation, i.e. following links. In full-fledged Web applications other kinds of user input can exist, e.g. entering values in forms. In an ongoing research, the Hera framework is being extended towards that direction incorporating other kinds of user interaction in the design methodology and in the underlying software suite [54].

2.5 Related Work

The growing interest in WIS resulted in several research efforts which have yielded a number of frameworks that support WIS design: RMM [59], OOHDM [99], Araneus [83], Strudel [32], WebML [25], UWE [70], XAHM [23], and XWMF [66].

Most of these approaches are model-driven and make a clear separation between the application domain description and the navigational aspects of WIS design. However, very few of these approaches explicitly consider adaptation and even fewer consider integration, as opposed to what is the case in our Hera framework\textsuperscript{7}. In this section we discuss in more detail RMM and OOHDM as representative examples of such systems.

2.5.1 RMM - Relationship Management Methodology

We start with a closer look at RMM [60; 59]—the predecessor of Hera. RMM is a methodology that covers the navigation design phase and uses the Entity-Relationship (E-R) approach for the application domain modeling. Similarly to Hera also RMM models the navigational aspects of a Web application by a so-called Application Model (AM). The AM consists of slices and relationships among them. Slices represent meaningful chunks of information that typically will be displayed within one page. Slices contain sets of attributes from one or more (related) entity types of the ER domain model. For instance, the slice \textit{Technique.Detail} in Figure 2.5.1 has the \textit{name} and the \textit{description} attributes from its root entity type \textit{Technique} and an index of \textit{name} attributes from the \textit{Painting} entity type. The relationships among the slices are of two types: aggregations used for slice nesting, and references creating (hyper)links between the slices. For presenting a collection of elements, slices can contain set access structures: \textit{index} for list access to multiple instances and \textit{guided tour} for sequential page-by-page access. We remark that the RMM methodology

\textsuperscript{7}Two notable exceptions which do consider integration are Ontowebber [61] and Seal [78]; we treat them in the related work section of Chapter 3 where we discuss the Hera integration framework.
does not explicitly cover the presentation design and thus the details regarding the presentation have to be hardwired by the programmer who transforms the application model into the rendering platform, e.g. HTML. RMM also does not explicitly support adaptation. However, it is still possible to manually build multiple application models based on the same domain model suiting different users or user groups/platforms.

A strong point of RMM is that the design of the application domain model is clearly separated from the application (navigation) model. The simplicity of the RMM methodology and its precise description proved to be a solid foundation for its extensions — the Hera methodology being one of them.

![RMM slice diagram](image)

Figure 2.18: RMM slice diagram.

### 2.5.2 OOHDM - Object-Oriented Hypermedia Design Methodology

The OOHDM methodology [99; 100] represents an object-oriented approach to WIS design. OOHDM captures the application domain in the form of a conceptual schema. The OOHDM conceptual schema is analogous with the UML class diagram; it uses classes with attributes and relationship constructs like inheritance aggregation, and association, to capture the semantics of a particular domain.
### 2.5. RELATED WORK

OOHDM models navigational aspects of the WIS design by the **Navigation Class Schema** and the **Navigation Context Schema**. The first one concentrates on the instances of which classes are used, whereas the second one concentrates on the ways how the instances can be visited.

The **Navigation Class Schema** contains navigation classes (nodes) derived from the conceptual classes by selecting and combining attributes from (possibly) different related conceptual classes. Navigation classes represent information intended for a certain audience and certain navigation context (there can exist more navigation views on the same conceptual model). Attributes in OOHDM are either of type data or of type (hyperlink) anchor. Figure 2.5.2 provides an example of the navigation class schema.

The **Navigation Context Schema** represents the navigation structure of the application and consists of navigation contexts. Navigation contexts are composed from the navigation classes, hyperlinks, and access structures (e.g. indexes, guided tours, indexed guided tours, menus) and represent collections of navigation classes instances that can be explored in some way (e.g. sequentially). Figure 2.20 shows an example of a particular instance of a painting technique: From a concrete technique one can navigate through the **Technique index** to the collection of instances of paintings exemplifying the technique.

Although the designer can achieve adaptation by realizing multiple navigation views (e.g. for different users or user groups) of a certain domain model, there is no specific support for building adaptation in OOHDM. However, there are some proposals for extending OOHDM with adaptation. The approach described in [100] includes a set of recommen-

---

#### Figure 2.19: OOHDM navigation class schema example.

<table>
<thead>
<tr>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>description: String</td>
</tr>
<tr>
<td>paintings: list(Anchor(P:Painting where P exemplifies Technique(self)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Painting</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>year: Integer</td>
</tr>
<tr>
<td>picture: Image</td>
</tr>
<tr>
<td>author: Anchor(P: Painter.name where P paints Painting(self))</td>
</tr>
<tr>
<td>technique: Anchor(T: Technique.name where exemplified by Painting(self))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Painter</th>
</tr>
</thead>
<tbody>
<tr>
<td>name: String</td>
</tr>
<tr>
<td>biography: String</td>
</tr>
<tr>
<td>paintings: list(Anchor(P: Painting where P painted_by Painter(self)))</td>
</tr>
</tbody>
</table>
OOHDM navigation context schema example.

OOHDM appears to follow the main lines of the object-oriented design approach and adds specifically the navigation (and presentation) design to the development process. Some modeling aspects are formally specified (e.g., temporal logic is used in the definition of contexts) in OOHDM. The conceptual model and the navigation model are clearly separated. While multiple navigational views are possible, adaptation is not explicitly described unlike in the case of Hera. OOHDM also does not cover integration and assumes that the data is all available from a single source.

Similarly to Hera, both RMM and OOHDM have their supporting tools for interpreting the design models. However, their underlying software tools appear to be rather monolithic and closed frameworks. Hera, on the other hand, can be seen as a set of collaborating components which exchange information in a (semantic Web) standardized fashion and it is possible to replace one component from the Hera framework with another (possibly third party) component, e.g., the use of the AMACONT presentation engine as an alternative for the Hera presentation layer [33].

2.6 Hera Summary

Modern Web information systems are data-intensive Web-based applications which automatically generate, as response to a user query, hypermedia presentations from data gathered from several different sources. These systems typically cover a number of aspects associated with this process: conceptual design, integration design, navigational design, and presentation design. Moreover, adaptation techniques need to be often deployed in various stages of the design process to deliver a suitable personalized presentation that matches the user needs. To support design and implementation of such complex systems, one needs both a methodology that guides the designer along the design process, and a software suite that realizes the design specification into a functional Web information system.

In this Chapter we answered Research question 1. Taking in consideration the Web evolution we proposed the Hera framework for the design of Web information systems that could be deployed on the Semantic Web platform. Such information systems make
use of Semantic Web technology to support Web application interoperability. The Hera framework includes a model-driven methodology covering two principal design phases: data collection and presentation generation. These phases consist of several design steps which result in the following specification models: conceptual model, integration model, application model, and presentation model. Each of these models covers different aspects involved in the design of Web information systems.

The Hera framework uses RDF(S), the Semantic Web (meta)data standard, to represent the individual design models. The semantic capabilities of RDF(S) are mainly exploited in the data collection, in particular for the integration and application domain modeling. At the same time, the RDF/XML model serialization enables the use of XSLT stylesheets as transformation specifications between the different model instances in the presentation generation phase. However, this approach can be satisfactorily applied only if one does not need to use the RDF(S) inference rules in the transformation specification, which was our case as these rules were resolved during the data collection phase.
Chapter 3

Hera Integration Framework

This chapter focuses on the integration aspects of WIS design. It presents the Hera integration framework consisting of a model-driven integration specification formalism and an underlying integration engine which executes the integration specification. The integration specification addresses the problem of relating (parts of) the source ontologies to our conceptual model. It provides the necessary primitives to establish at schema level virtual semantic channels between source concepts and their relations on the one hand, and their CM counterparts on the other. This specification then serves as input for the Hera integration engine, which establishes at instance level the connection between the sources and the Hera application. For this purpose we adopted a mediator architecture based on the on-demand data retrieval paradigm which assures that the gathered data is always up-to-date.

The chapter is structured as follows. Section 3.1 looks at the context introduced by Hera for which we need to devise an integration solution. The section states our assumptions about the sources which data is to be integrated and identifies a number of requirements for the integration process. Section 3.2 introduces the Hera integration framework, outlining the general integration architecture adopted by Hera. Section 3.3 proposes the Hera integration model, describing its vocabulary and the integration modeling primitives. Section 3.4 describes related work in the field of data integration, particularly focusing on approaches tailored for Web engineering. Finally, Section 3.5 concludes this chapter with a brief summary in which we check whether the proposed integration framework meets our requirements.

3.1 The Hera Integration Context

We focus on the integration of sources that are capable of exporting their schema in an RDFS based ontology and of exporting their data upon request (e.g. a query) in RDF. In other words, we assume that each source offers its data on the Semantic Web platform providing RDF query services. We argue that this is a plausible assumption in the sense that our main target is the development of Semantic Web information systems, and
choosing the RDF(S) platform as a (minimal) requirement for our data sources follows the Semantic Web strategy to have RDF(S) as its foundation. In case some sources do not meet this requirement, i.e. they do not provide their data in RDF(S), a layer of wrappers which would reconcile the syntactic differences might be needed [121].

The sources that we consider are typically autonomous and we cannot impose structural changes in the way they organize their data. The sources can have their data semantically structured in different ways, which is reflected by the heterogeneous RDFS ontologies that describe them. We assume that the source ontologies are self-contained in the sense that a source ontology completely describes the data at a particular source, and that the sources are independent from each other, i.e. there is no cross-referencing among source ontologies. With this assumption, the local inferencing of sources combined with the inferencing at the mediator guarantees completeness of the result.¹

The self-containment assumption does not exclude sources that provide similar data under similar or even identical ontologies. On the contrary, it is often the case on the Web that the information is duplicated and offered (possibly with a different flavor) from several sources. We can (virtually) group such sources into semantically-close clusters; sources within a cluster do not necessarily have the same structure but should provide approximately the same semantic content. Within a cluster, the sources can be dynamically ordered based on several notions of quality depending on the user context.

Similarly to sources, also the conceptual model of our application is considered autonomous with respect to the integration process. As explained in the previous chapter, the CM is designed beforehand and represents more than just a straightforward union of data coming from different sources. The CM captures a common semantic agreement within a certain domain and the integration framework is not allowed to change it. The role of the integration framework is to reconcile the differences between the sources and the CM, so that the CM can be filled with source data.

### 3.1.1 Hera Requirements for Integration

By analyzing the specifics of WIS and the context of the Hera framework in particular, we formed the following set of requirements for the integration framework, affecting both the integration formalism and the integration engine that executes it. The existing approaches to data integration, e.g. [116], often do not meet some of these important prerequisites, which renders them less suitable for designing WIS.

1. **The use of (open) WWW standards**

   While in the past the main consumer of the information provided by WIS was usually a human, currently there are more and more (Web) applications that need to process this information as well. Moreover, WIS themselves often consist of a complex composition of collaborating (Web) components, such as the components in the Hera software suite.

   ¹If this assumption is not satisfied and completeness is still a desired property, the integration designer has to explicate the schema dependencies by means of the integration specifications.
This requires the use of Web standards throughout the entire process, including the data collection phase. To promote the re-use of integration specifications by other parties on the Semantic Web, it is useful that not only the information to be integrated, but also the integration specification itself is expressed in the same description standard.

2. Ontology level, model-based approach

Since WIS are data-intensive applications with large numbers of instances per concept, it is not feasible to specify integration mappings for every individual instance. This data-intensive nature combined with the separation-of-concerns principle adopted by Hera implies that the integration should be expressible in an explicit model which reasons in terms of source schemata/ontologies rather than in terms of the actual data instances. The instance integration has to be realized on-the-fly during the query evaluation, based on the integration model which is established at ontology level.

3. Independence of the conceptual model

As argued above, the conceptual model is assumed to be developed independently from the source ontologies by the application designer (possibly with a team of domain experts). It reflects the agreed conceptualization of a domain of interest usually within a certain information community for which the Web information system is intended. Moreover, in a Web engineering framework such as Hera, the CM often serves also as an interface between different collaborating components (in Hera’s case it is the mediator and the presentation generation engine), and thus must be maintained with care. Sources targeted by Hera are typically autonomous and their underlying ontologies can change. It is undesirable to propagate these changes to the structure of the conceptual model which has a meaning on its own and it is not just a reflection of what the sources can offer. The integration model has to shield the conceptual model and thus also the rest of the framework from changes at the source side.

4. Expressivity of the integration formalism

The integration formalism must be able to cover a wide range of semantic heterogeneities frequently occurring on the Semantic Web. This includes schema heterogeneities where the source ontologies can differ in their concepts, properties, and structure. In particular, RDFS models often contain sequences of connected properties—so-called ontology or schema paths. These paths can differ considerably among the sources and the conceptual model. It is essential that the integration formalism is able to reconcile such path discrepancies.

The identity of the RDF resources on the Semantic Web is thought to be established by the notion of a so-called Uniform Resource Identifier (URI). We observe however that the URI only works as a unique identifier in controlled environments such as corporate intranets or bank information systems, where a higher authority can impose the naming conventions. This does not always hold for the autonomous sources on the Web that Hera targets as...
potential content providers. Different sources can use different URIs for referring to the same real world objects; this is referred to as designation heterogeneity and the integration model must be able to address this issue at ontology level. To reconcile these designation differences is an essential prerequisite to the successful evaluation of complex queries that involve the joining of data coming from multiple sources. Such complex queries occur frequently in Hera, as even a simple user information request may trigger the retrieval of several related data elements which are used to produce the (surrounding content of the) hypermedia presentation.

5. Flexibility of query evaluation

While the integration formalism must be expressive enough to capture the implicit knowledge about sources in an explicit form, the query evaluation has to be flexible with respect to the user needs and deliver the right data in the right order. In particular, when dealing with multiple sources offering similar content, e.g. instances of the same concept in the conceptual model, the order in which these sources are consulted should not be hard-wired in the integration engine, but programmable by means of the integration model and adaptable to the user context. For instance, it may be desirable to consult a source with low-resolution images when the user is using a PDA, while a different source with high-resolution pictures should be preferred when the user browses the presentation on a PC with a broad-band connection.

6. Freshness of Data

Hera has the ambition to support the engineering of WIS which often use as content providers other autonomous Web sources. Both the structure and the data of these sources may change without notification. While the structural changes are assumed to be less frequent, the actual data can change frequently. In this context the freshness of the gathered data is an important aspect and should be guaranteed.

3.2 Hera Integration Framework

The main purpose of the integration framework is to address the above requirements and to provide a semantically unified interface for querying (selected) heterogeneous information sources that populate on request the conceptual model with data. The integration framework covers the integration design phase which is materialized in a so-called integration model, the specification of which is used by the underlying integration engine.

As opposed to database schema integration [116], we do not aim at merging all possible source concepts together to provide a cumulated view of all attributes. We argue that such approach offers weak semantics, where the understanding of the semantic structure of all integrated sources is effectively left up to the user who is issuing the query against such a view. Instead, the Hera approach advocates the use of a conceptual model which is
3.2. **HERA INTEGRATION FRAMEWORK**

tailored to the user needs (from the semantical point of view) and connected to carefully selected parts of the underlying sources by means of our integration model.

An important task in this process is to choose the right concepts occurring in the source schemata and to relate them to the concepts from the CM. This task is also referred to as ontology mapping, and it is in many ways similar to (or part of) the process of aligning or merging ontologies. At this stage, when the machine processable semantics on the Web does not accommodate all the implicit semantics that humans associate with data, it is widely acknowledged that to fully automate this process is unfeasible. In [62] the authors surveyed a number of state-of-the-art ontology mapping frameworks, none of them being able to perform fully automated mapping between ontologies. Some frameworks completely rely on the human insight to create such mappings, while other adopt a so-called semi-automatic approach. The main idea is to find some initial correlations between the concepts from different ontologies, leaving the actual creation of the final mapping descriptions on the designer.

The approaches for finding the initial correlations are usually based on lexical matches, relying mostly on dictionaries to determine synonyms and hyponyms. However, this is often not enough to yield good results. In [90] the structure of the ontologies is also taken into account when searching for corresponding concepts. Neither this approach delivers satisfactory results, especially if the ontologies are constructed differently, e.g. having a very different structure or differing in the depth of the class hierarchy. This is often the case in uncoordinated development of ontologies across the Web, and that is why ontology mapping is currently mostly a manual process, where a designer identifies the concepts that are similar and records the mappings between them.

For Hera, the data collection is just a beginning of a pipeline of data transformation steps at the end of which we produce a full-fledged hypermedia presentation. Every mistake in the mapping phase would propagate (and magnify) to all other steps in the pipeline. In this context, also the Hera integration framework currently relies on the insights of the designer who creates the integration model (as well as all the other models in the Hera framework).

Because of the Web nature of our target applications in combination with the dynamically changing data in the underlying sources, we chose to base our integration framework on the on-demand retrieval paradigm [76] assuring that the delivered data is always up-to-date. The notion of mediator as introduced in [124] laid the foundation for building mediating architectures that have the ambition to overcome the semantic heterogeneity and facilitate a seamless access to data coming from many heterogeneous sources. In order to provide software support for the data collection phase, our prototype implements such a mediating architecture. Figure 3.1 depicts the Hera integration framework. The main component is the mediator, which uses the integration specifications in the form of an integration model to gather data from multiple sources, reconciling the semantic heterogeneities of the sources with respect to the conceptual model.

In the next section we mainly focus on the integration design, explaining the specifics
3.3 Integration Design

The task of designing integration support for WIS engineering is rather complex. In general, it includes steps like finding relevant information sources, parsing and identifying their content, and specifying the semantic bridges between the relevant parts of the sources and the conceptual model. In our research we primarily focus on the last activity: We consider the reconciliation of the semantic heterogeneities between the sources and the conceptual model the primary challenge for the Hera integration framework. In order to address this issue we propose an integration model formalism.

As we stated in our requirements, it is beneficial if the integration formalism can be expressed in the same data format as the actual data that is integrated. It both facilitates the semantic interoperability and allows for reasoning about the integration phase at a higher level of abstraction. The integration model vocabulary defines in RDF(S) syntax the integration primitives such as ontology path, articulation, and decoration. In the process of describing the vocabulary of our integration model we mainly rely on the graphical syntax in order to depict the RDF(S) expressions such as rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range, etc. The complete RDF(S) translation of the integration model...
vocabulary is provided in Appendix A.

Next, we introduce the main concepts of the integration model vocabulary and illustrate their use.

Ontology Path

The basic component of our integration model is that of an ontology path as depicted in Figure 3.2. It basically consists of a sequence of nodes and edges, where the nodes represent concepts from a particular ontology and edges indicate the nodes’ properties, either concept relationships or (at the end of the path) concept attributes. The existing properties may be traversed either forward (the follow connector) or backwards (the backtrack connector). This allows for the use of source properties that are inverse with respect to their CM counterparts. Figure 3.3 shows an example where we have to traverse the source property paintedBy backwards (using the backtrack connector) in order to align it with the corresponding CM property. For this example we assume that the CM only contains the paints property.
The starting node and the ending edge of an ontology path are explicitly registered and may be associated with a processing instruction. Processing instructions are invoked by the mediator during the query evaluation. They typically perform the needed data transformations such as unit conversion, concatenation, or splitting of the values. In addition to a set of predefined processing instructions, the Hera framework also supports a dynamic linking of custom-defined Java-based processing instructions.

The starting node of every ontology path is a so-called PrimaryNode which represents the main concept the path is describing. In order to address the designation heterogeneity we provide the following kinds of identification mechanisms: idByURI, idByValue, idByList, and idByProperty. In the following we describe these identification approaches in more detail.

The idByURI identification indicates that the resources of that particular concept are identified by their URIs. This way of identifying assumes that all the sources that are integrated use the same URI naming scheme, or in other words there is no designation heterogeneity among them. As mentioned earlier, this can be guaranteed in controlled environments, like corporate information systems. Figure 3.5 shows an example where all the instances of the concept Painter are identified by their URI.

When the sources do not adhere to the same (instance) URI naming scheme, we turn to value identification denoted by idByValue. This property points to a concept attribute which contains the identification value. While identifying by URI follows the spirit of intensional RDF semantics (and we use it as default, i.e. when no identification is specified we assume idByURI), identifying by value is more in the spirit of the extensional semantics,
which is in a way a restricted version of the former. However, we argue that the limitations
of this pragmatic approach (such as the impossibility to distinguish between two different
resources that have coincidentally the same value of the identification attribute) are out-
weighed by the ability to perform join queries across sources that use completely different
URIs to refer to the same entities. We remark that identification by value is quite common
in the area of (relational) databases where tuples are considered the same if they have the
same values. Figure 3.6 shows an example where the instances of the Painter concept are
identified by the name attribute.

The main problem with value identification is that in many cases a single attribute does
not have to guarantee uniqueness. This often depends on the domain context, for instance
the name attribute may serve reasonably well for uniquely identifying classical painters
of a certain era, as they are not so many and name duplicity is not very likely to occur.
However, if we want to also include in our system modern painters and maybe other artists
as well, the chances are that sooner or later there will be two painters/artists bearing the
same name, which would be considered by the integration system as the same person. To
dercrease the likelihood of such errors the designer can select several attributes, which when
combined act as an identifier. This grouping is realized by the idByList feature, which is a
container where the designer enumerates the attributes needed for the identification. These
attributes are sometimes also referred to as discriminating, as they discriminate based on
their value different entities. Figure 3.6 shows an example where instances of the Painter
concept are identified by the combination of the name and the dateOfBirth attribute.

In general, the more discriminating attributes are included in the group, the smaller
the probability of misidentification is. On the other hand, the more such attributes are
used, the higher the performance penalty is, as they all have to be examined during every identification comparison. Typically, not all discriminating attributes contribute to the uniqueness of a concept to the same extent. For instance the family name (usually) discriminates more than the date of birth. The extent of this contribution can be determined by sampling on a training data set as suggested by [49]; this approach helps with identifying the discriminating attributes.

In some cases, neither the URI nor even the (direct) concept attributes are sufficient to establish the identity. In its essence, RDF(S) is a data model with “flat” literal data types. If the designer wants to combine the literal types into a (possibly nested) record-like structure, he needs to create an “artificial” concept which holds the grouped attributes. For instance, to model the attribute name as being composed from the literal attributes first name and last name, one needs an intermediate concept e.g. Name which groups these two attributes. In order to be able to identify a concept by attributes which are “further away”, the integration model includes the idByProperty feature. The idea is to traverse recursively designated identification properties, until either idByUri, idByValue, or idByList is reached. Figure 3.7 shows an example where the personal data of the Painter (e.g. the name and the dateOfBirth attribute) is grouped under the concept Bio. In order to identify instances of the Painter concept, we first have to traverse the personalData property and then resolve the list identification as explained above.
3.3. INTEGRATION DESIGN

Articulation

The notion of ontology path as described above serves as the basic component for creating mappings in our integration model. We refer to these mappings as articulations as they articulate parts of the source ontology in terms of the conceptual model. Figure 3.8 depicts the part of the integration model vocabulary which defines Articulation. In essence, a basic articulation is a pair of two ontology paths: the To path pointing to the conceptual model, and the From path pointing to a source. The idea is that in order to populate the data for the primary concept of the To path we need to ‘execute’ the From path on the source. The From path can be seen as (a part of ) the query which the mediator routes to the source when making the data retrieval request. From this perspective, articulations serve as translation rules which translate ontology path expressions from the conceptual model to the appropriate ontology paths in sources.

Figure 3.9: Source ontology.
To illustrate the use of articulations, we consider the conceptual model introduced in the previous chapter in Figure 2.3 and a source described by the ontology depicted in Figure 3.9. The task is to populate the Painting concept and the paintedBy concept relationship from the conceptual model. Looking at the source ontology, we observe that the path consisting of the creationDetails and author relationships corresponds to the paintedBy relationship in our conceptual model.\footnote{In order to make these kind of observations, the designer needs to understand the intended semantics of the source ontology; this frequently involves analyzing the surrounding context which is often not available in a machine processing format.} Figure 3.10 depicts the articulation that relates the two corresponding paths.

![Diagram](http://www.win.tue.nl/~Hera/ha/cm)

**Figure 3.10:** Articulation example.

To augment the expressive power of our integration framework, the integration model vocabulary also provides a means for expressing transitive articulations. A transitive articulation allows for expressing recursive path definitions. To illustrate this, consider an example of a simple conceptual model (Figure 3.11 - top left) with the Employee concept and the superior concept relationship. This relationship connects two employees A and B if B is higher in the managerial hierarchy of the organization than A. However, the source from which we extract the data (Figure 3.11 - bottom left) provides only the information about one level of the managerial hierarchy at a time, e.g. an employee B oversees A as its (direct) inferior. To reconcile this semantic heterogeneity, we use a transitive articulation (Figure 3.11 - right). This articulation prescribes that in order to retrieve the instances of the superior relationship for a particular employee in the CM, the oversees source relationship has to be traversed backwards and recursively.

We remark that the transitive articulation primitive extends the expressive power of RDF(S). In our integration model vocabulary we used the owl:TransitiveProperty class to...
indicate the transitive behavior of *TransitiveToEdge* in the articulation definition. The transitive articulation should be used with care as it potentially degrades the performance of the query evaluation. At the end of this section, we discuss further the tradeoffs between the expressive power of the integration model and the performance of the query evaluation.

![Diagram](http://www.wis.win.tue.nl/~Hera/cm)

**Figure 3.11: Transitive articulation example.**

### Decoration

While articulations connect the relevant parts of the source ontologies to our conceptual model, many times it is useful or even necessary to explicate some source properties and (a part of) the implicit context which surrounds the source. This knowledge is usually acquired by the designer during the process of articulating the source ontology paths in terms of the CM. We argue that when made explicit, this knowledge can enhance the expressive power of our integration model and, when properly taken into account, improve the performance of the integration engine. For this purpose we introduce the notion of *decoration*. A decoration explicitly captures the implicit knowledge of the designer, and can be either applied to a source as a whole, or to a single articulation. In the former case, the decoration affects all articulations describing that particular source; in the latter, the decoration concerns only the particular articulation, possibly overriding a related decoration at source level. Figure 3.12 introduces the part of the integration model vocabulary which defines the notion of *Decoration*. As depicted in the figure, decorations are of three types: *Rank*, *Mirror*, and *Restriction*. 
The *Rank* type allows to assign different quality dimensions to sources (or individual articulations) and rank sources/articulations by specifying values for these quality dimensions. In this way, the designer can for instance specify that one source contains images of higher quality than those contained by another source, which in turn may be ranked higher for the bandwidth. Some of the quality dimensions (mainly those which deal with technicalities such as the bandwidth or the processing speed) can be determined and changed during the runtime by a software component that monitors the sources. The mediator is able to take these changes into account, which improves both the flexibility and the performance of the query evaluation.

While decorating measurable dimensions such as bandwidth can be easily automated, ranking sources for the quality of their content is mainly a task for the designer. In the context of the sources targeted by Hera, we however assume that content reputation is usually less volatile than the technical qualities. For instance, the processing speed may change rapidly depending on how many clients are trying to access the source at a given moment.

The quality dimensions introduced by the *Rank* decorations can be seen as an extension of the conceptual model, in the sense that they may be used in the queries issued by the user or generated by the application layer. This way, the mediator takes into account the user context, producing a flexible query plan depending on the current user preferences.

Another decoration class is that of *Mirror*. It addresses the case when the sources (or just some of their articulations) are replicated or completely substitutable by other sources. A source or an articulation that is designated as a mirror of another source/articulation is considered equivalent by the mediator and the query can be routed to either of them. This information is particularly useful to recover from a source failure, but it can also be
used to provide a better load balancing (taking into account also the Rank decorations).

![Diagram of Restriction decoration example]

**Figure 3.13:** Restriction decoration example.

Restriction is the third type of decoration that we define. As its name suggests, it serves for explicating (implicit) restrictions over sources or articulations. A restriction is expressed as a pair consisting of an ontology path (either from the CM or from the source) and a restriction value, and asserts that the restriction value and the value of the path attribute are equal.

If the path in the restriction points to a CM attribute, the restriction decoration indicates that the value of that particular attribute is constant for a given source/articulation. For instance, suppose a source “dutch-painters.nl” which contains data (only) about painters form the Netherlands. Given its (implicit) national context, it is natural when such a source does not explicitly mention the nationality attribute in the source ontology (as it is clear that all painters are Dutch). If however our conceptual model contains the nationality attribute of a painter, the designer has to use a restriction for that particular source. This restriction contains the path to the nationality attribute in the CM and the value “Dutch”. Based on this, the mediator sets the nationality value for every painter instance that is retrieved from this source.

Another use of restrictions is when the restriction path points to the source. The restriction value then indicates a selection condition over the instances from that source/articulation. For instance, let us consider the source described in Figure 3.9. The articulation from Figure 3.10 links the Painting concept from the conceptual model to the MasterPiece concept from the source. This specification is correct if the source lists only painting masterpieces.

\[\text{If needed, the designer can override the default comparison on equality (\(=\)) by specifying other comparison operators (e.g. } \leq, \geq \text{) in the prefix of the actual restriction value.}\]
However, if the source also contains other kinds of art such as statues or photographs, this articulation may yield wrong instances. In order to handle this heterogeneity, we need to restrict it to return only those masterpieces that are paintings. One way of doing it could be to discriminate the masterpieces based on the \textit{materialUsed} attribute. For simplicity, let us assume that all paintings used “canvas” and that there are no other kinds of masterpieces on this material. Figure 3.13 depicts a restriction decoration for this example. This restriction applies to \textit{articulation1} defined in Figure 3.10 and limits this articulation only to those masterpieces that used “canvas” as their material.

\textbf{Notes on the Integration Model Design}

The integration vocabulary presented in this chapter consists of articulations augmented by decorations, and provides an expressive declarative integration language which may be further enhanced by custom-defined processing instructions. While, as mentioned above, these processing instructions are meant for simple value-oriented transformations e.g. unit conversions, they may also implement fairly complicated translations or reasoning strategies e.g. utilizing external data translation indices or dictionaries. One should note that the more integration intelligence is hardwired into these java-based processing instructions, the less declarative the integration model becomes. We recommend to keep these external processing instructions as simple as possible, in order to preserve the declarative nature of the integration model and also the performance of the query evaluation which is inevitably affected by the way these instructions are implemented.

In general, we can conclude that there is an inherent tradeoff between the expressiveness of the integration model and the performance of the underlying integration engine. The more expressive integration primitives introduced in this chapter impose a higher performance penalty compared to their less-expressive counterparts. For instance a URI-based identification is (slightly) less expensive than a value-based identification, which in turn is less expensive than an identification by list. Another example is the transitive articulation which can be substantially more expensive than the ordinary articulation. It is up to the designer to consider which primitives are the most suitable for the application, and whether the performance penalty introduced by them is acceptable.

\section{3.4 Related Work}

The interest in data integration is a reoccurring research theme in several areas of computer science. Researchers and practitioners from databases, information systems, and AI produced a large body of research related to this issue. In this section we briefly review related work from focusing on view definition languages devised in the database area and detailing two related approaches from the field of Web-engineering.
3.4. RELATED WORK

View Definition Approaches

The integration formalism proposed in this chapter can be compared to source description languages used in database schema integration, as the problem of ontology mapping is often regarded as being similar to the schema integration problem in the database area [62]. A number of source description languages were proposed in the context of this research, the main representatives being Local-As-View (LAV) [75], Global-As-View (GAV)[44], and their combination called GLAV[43].

The LAV [75] approach describes the sources as view over the mediated data. The LAV source descriptions have the following form:

\[ s(X) \Rightarrow m_{1}(X_{1}, Z_{1}) \land \ldots \land m_{k}(X_{k}, Z_{k}) \]

Where \( s \) represents a source relation, \( m_{i} \) is a relation from the mediated schema. Further, \( X = \cup_{i} \bar{X}_{i} \) represents a tuple of universally quantified variables and \( Z_{i} \) stands for existentially quantified variables. Every variable takes a value from a particular attribute domain to which it is assigned by its position in the relation. LAV descriptions are able to handle cases where the mediated schema contains details which are not present in all sources. The disadvantage of the LAV approach is that the mediated schema has to use all the attributes shared by multiple source relations.

The GAV [44] approach describes the relations of the mediated schema as views over the source relations. The GAV source descriptions have the form:

\[ s(X_{1}, Z_{1}) \land \ldots \land s(X_{k}, Z_{k}) \Rightarrow m(X) \]

Unlike in the case of LAV, GAV descriptions are able to handle cases where the sources contain details that are not present in the mediated schema. On the other hand all mediated schema relations must be present in the sources. In other words, the mediated schema can be subjected to changes depending on the existence of the source relations. As advocated earlier, this is very undesirable for the Hera conceptual model as the CM both represents an engineering artifact of capturing the application domain and serves as an interface for the presentation generation module.

The GLAV [43] approach combines the above two sources description languages. The GLAV source descriptions have the following form:

\[ S(X, Y) \Rightarrow m_{1}(X_{1}, Z_{1}) \land \ldots \land m_{k}(X_{k}, Z_{k}) \]

\( S(X, Y) \) is either a conjunction of source relations or a query predicate of a datalog query over the source relations. GLAV combines the expressive power of LAV and GAV. Similarly to the transitive articulation of our integration model, GLAV also allows for expressing recursive mappings which are useful for expressing arbitrary long paths. However, even the GLAV approach fails to address the possible designation heterogeneity of
instances and the flexible ranking of sources depending on the user context.

In the rest of this section we focus on WIS design frameworks, particularly those that provide support for the integration process. As mentioned earlier, despite the fact that the integration is an important aspect of WIS design, many Web engineering approaches do not address this issue explicitly in their methodologies, as opposed to what is the case in our Hera framework. Two notable exceptions which also cover integration are Ontowebber [61] and Seal [78]. Like Hera, both of these approaches are centered around RDF(S) which facilitates their deployment on the Semantic Web platform.

OntoWebber

OntoWebber [61] is a system for building and managing data-intensive Web sites. Similarly to Hera, it adopts a model-driven ontology-based approach for declarative Web site management and data integration. It advocates the use of ontologies as the basis for constructing different models necessary for WIS design. It clearly separates the WIS design into domain modeling, navigation modeling and presentation modeling, and also offers (a limited) personalization support.

OntoWebber supports the integration of heterogeneous data sources based on RDF as common format for modeling semistructured data. In the first step OntoWebber focuses on syntax reconciliation converting all source data into RDF. This RDF data (both the schema and the instances) is replicated and stored locally. From this point of view, OntoWebber acts as a data warehouse and does not guarantee the freshness of its data, as opposed to the Hera integration framework which implements the on-demand retrieval paradigm, assuring that the retrieved data is always up-to-date.

After the replication phase, the local copies of the source data are articulated by the designer in terms of a reference ontology which captures the domain of interest. Unlike our integration model, OntoWebber articulations focus mainly on naming differences, failing to reconcile larger ontology discrepancies such as those described in the examples in Figure 3.10 and 3.13. As a consequence, OntoWebber does not facilitate those queries that include the joining of data that originates from different sources.

SEAL

The Semantic PortAL (SEAL) approach [78] offers a conceptual framework both for information integration and Web site management. It proposes a number of design models, separating the domain description, navigation design and personalization. Similarly to Hera, the application domain is captured into an ontology which serves as a semantic backbone for the rest of the framework; this ontology is expressed in RDF(S).

The integration phase of SEAL mainly focuses on converting the different data formats of the sources (e.g. relational databases, HTML, XML) into a common datamodel expressed in RDF(S). After the syntactic reconciliation, the source data is locally replicated. Similarly to OntoWebber, also SEAL implements the data warehouse approach which ren-
ders it less useful for the autonomous and dynamic sources that are targeted by Hera. The integration language of SEAL consists of a number of mapping rules [108]. These rules mainly cover the issues related to the reverse engineering of databases which use different underlying data models. The SEAL integration framework does not address the designation heterogeneity problem, assuming that the referencing mechanism is consistent across the integrated sources.

3.5 Integration Framework Summary

In this chapter we focused on the integration aspects of Hera, addressing our second research question, as introduced in Chapter 1.

After analyzing the specifics of WIS and the context of the Hera framework in particular, we identified a number of requirements for our integration framework. Here we briefly summarize the proposed integration formalism relating it to the identified requirements.

To address Requirement 1 we express our integration formalism in RDF Schema—the W3C standard for defining vocabularies on the Semantic Web. This facilitates not only the reuse of data but also the integration specification.

The proposed integration formalism takes the form of an (RDF) model which maps the concepts and relations of the Hera conceptual model into those from the source ontologies. The instances are reconciled on-the-fly during the query resolution. This ontology-level mapping approach follows the data-intensive nature of WIS, addressing Requirement 2.

To address Requirement 3, the integration model effectively decouples the external sources from the rest of the Hera framework. When a source changes, i.e. its ontology is altered, this change is reflected only in the articulation(s) concerning this source, while the conceptual model stays intact, preserving its independence.

In our integration model, the articulations are able to reconcile different ontology paths. The articulation component also addresses the designation heterogeneity by providing different primitives for the identification of instances; this allows for queries that involve joining of data coming from different sources. When needed, articulations may be augmented by restriction decorations which explicitly capture the implicit articulation context. A combination of all these elements addresses Requirement 4.

To address Requirement 5 we propose ranking decorations, in which the designer explicates the otherwise implicit properties of the sources. These decorations are taken into account by the mediator during the query answering process, allowing a flexible query evaluation which considers the user context.

Finally, Requirement 6 is addressed by the architecture we adopted for our integration engine. We proposed the use of a mediator architecture which implements the on-demand retrieval paradigm, where data is collected on-the-fly from the sources during the query evaluation process. This guarantees that the retrieved data is always up-to-date.
Chapter 4

Distributed RDF Query Processing

To discuss the technical aspects of distributed RDF query processing we first need to detail the characteristics of our distributed setting, where the relevant information might be scattered over several sources. This requires a suitable architecture where the query-answering system locates relevant information, retrieves it, and combines the individual answers. This chapter proposes several mediator architectures for distributed access to RDF data, identifying their main features with respect to RDF query processing and optimization.

Next, the chapter concentrates on details of the distributed RDF query processing. We focused mainly on intelligent query routing where the sub-paths of a particular query need to be distributed to the appropriate sources. We proposed a path index data structure which allows for efficient access to the mapping information form the integration model introduced in Chapter 3. Using this data structure we devised an algorithm for distribution of RDF path queries.

This chapter is structured as follows. Section 4.1 details several mediating architectures that implement the on-demand retrieval paradigm, discussing their basic assumptions and implications. Section 4.2 presents various index data structures and algorithms to support the localization of relevant data during query processing. Section 4.3 reviews the related work on distributed RDF architectures and index structures for object-oriented databases. Section 4.4 concludes this chapter with a short summary.

4.1 Integration Architectures

Considering the Hera integration requirements, the main attributes of our distributed RDF framework were defined in the previous chapter. We concluded that our framework should implement a mediator architecture which employs the on-demand retrieval paradigm in order to retrieve (upon the query request) up-to-date results. In this section we detail several mediator architectures which can be covered by our prototype, explaining the implications that they introduce for distributed RDF query processing.
4.1.1 Basic Mediator Architecture

The basic mediator architecture is described in Figure 4.1. The underlying repositories contain the RDF data instances that are to be retrieved for the conceptual model. These repositories are assumed to provide RDF query facilities, i.e. deliver their instances upon the query request. In our prototype, we implemented the repositories as stand-alone Sesame servers [18], but the ideas of our research are applicable for other local RDF storage and querying systems as well.

![Figure 4.1: Basic mediator architecture.](image)

The central component of this architecture is the mediator [124]; its main task is to reconcile at run-time the semantic heterogeneities of the underlying sources and provide a transparent access to their data. During the initialization phase, our mediator takes the integration model as input and transforms it into a look-up path index, which is used during the query execution.

The sequence of steps involved in the query processing is depicted in Figure 4.2. Given the query, the mediator needs to locate the right sources that contain the answer, split the query accordingly and distribute it among the selected sources. After the results come back, the mediator needs to combine them in an efficient way. The first part of the query answering process requires a suitable data structure for path indexing; this is addressed in Section 4.2.1. The local processing of results involves mainly a sequence of join operations, the optimization of which we detail in Section 5.1.4.
4.1. INTEGRATION ARCHITECTURES

In this architecture, the mediator tries to maximize the sub-queries that are being pushed to the underlying repositories. The reasons for that are threefold. First, unlike the mediator which maintains only an ontology-based path index, the source repositories contain instance-based indices which allow them to perform local optimization and thus provide for more efficient path query evaluation.

Second, the repositories can be seen as a parallel machine which processes the assigned queries. The higher the degree of parallelism (i.e. the local query processing), the less work is left for sequential processing at the mediator.

Third, the selective fragments of the query (i.e. the constituents of the Where clause), when pushed down, prevent the retrieval of those instances that do not fulfill the filtering condition and would be discarded anyway. This effectively reduces the traffic costs which often contribute largely to the overall response time.

4.1.2 Hierarchical Mediator Architecture

The rationale behind the basic mediator architecture is to hide from the user (or a system built on top of the mediator) the fact that the information is distributed across different sources. The user should not need to know where a particular piece of information resides: the system should behave as if all the information was available in a single source represented by the mediator. This perception of the mediator as a single source also leads to an extension of the architecture, where the mediated sources can be mediators themselves and in turn hide a set of distributed sources. This composition results in a hierarchical mediator architecture (HMA) as depicted in Figure 4.3.
This hierarchical structure of distributed mediators often mimics the hierarchical distribution of data in the real world. It allows Web information systems such as digital libraries or cultural heritage portals to change their scope from local to regional or national. Clearly, interoperability at this scale needs the solid foundation of frameworks like RDF(S) and the description formalisms built on top of these to provide the necessary expressiveness and flexibility required for the integration.

The sequence of steps involved in the query processing for this architecture is depicted in Figure 4.4. The query processing starts at the mediator which receives the query (*Mediator1*) and proceeds recursively with its subordinate mediators. Regarding the communication with the local sources the sequence is identical to that in Figure 4.2.

Both distribution architectures discussed so far have some limitations with respect to scalability and efficiency. The problem lies in the fact that the information sources offer only a limited functionality that concerns querying their own data. The joining of all partial results must be executed by the mediator. This centralization of the query processing is problematic because the mediator constitutes a single point of failure and the communication costs for sending large result sets from individual information sources can be a major performance bottleneck.
4.1. INTEGRATION ARCHITECTURES

Figure 4.4: Query processing message sequence, hierarchical mediator architecture.

Figure 4.5: Possible problems in a mediator hierarchy.

The extension of the basic architecture where some of the sources can be mediators themselves addresses some of these problems. If the central mediator fails it is still possible to access (some of the) data at the lower levels of the hierarchy through the subordinate mediators. From the performance point of view, the mediators at the lower levels take over
parts of the processing. In particular, they process the sub-query sent to them thereby reducing the amount of data that is extracted from the underlying data sources.

However, the hierarchical mediator architecture introduces some problems of its own as can be seen from Figure 4.5. When an information source is indexed and monitored by more than one mediator (Figure 4.5 A), inefficiencies are introduced. A query that refers to the common source will be answered by the two mediators that index it, resulting in duplication of retrieval and communication effort. Another flaw in the construction of the hierarchy arises when a mediator indexes one of its ancestors (Figure 4.5 B). This problem is more severe as it essentially causes a deadlock state (or an infinitely recursive query). In order to prevent these issues, care has to be taken when attaching sources (especially other mediators) to the system. In our prototype, we maintain a pure hierarchical structure of the framework by performing a safety check for loops when the source is being added to the system. In order to be able to do that, the integration model distinguishes between simple sources and those sources that are themselves mediators (see the IM vocabulary in Figure 3.2).

### 4.1.3 Cooperative Mediator Architecture

The architectures discussed above were all data-centric in the sense that the communication between two related nodes in the system was strictly data-driven, with a simple protocol consisting of a request (query) followed by a response (RDF data). As a consequence, the processing of partial results had to be always performed on the upper node in the hierarchy. This in turn was the major cause of the disadvantages listed in the previous section.

In order to address these issues, a more flexible architecture is needed. In such architecture, (some) mediators should agree to cooperate by offering processing capabilities, especially the join evaluation, to other nodes in the network. To facilitate this functionality, the software stack of the mediator has to be extended with a new communication layer. This layer allows for requests of the type “join the attached data with the data obtained from your sources and deliver the result to mediator X”.

Figure 4.6 depicts such a cooperative mediator architecture (CMA). The solid directional lines indicate the usual data request/response channel (the direction points towards the source node), while the dotted lines represent the newly introduced processing channels which connect communication layers of the cooperating mediators. The CMA is a natural evolutionary step from the approaches described in the previous sections, since it still maintains the simple data request/response communication but the evaluation of the partial results can be distributed over the cooperating mediators.

While the hierarchical mediator architecture described earlier is more of a client-server nature, the cooperative architecture moves us closer towards the P2P paradigm. A mediator that implements the communication layer can be considered as a peer that offers its processing capabilities to some of the other peers.
However, unlike in a typical P2P system where the number of peers is likely to be quite high, our target applications, such as digital libraries or internet portals, do not typically reach millions of collaborating nodes. The growth of such network is usually much more controlled and also peers are less volatile than in a typical P2P system where peers join and leave the network in a very dynamic fashion. Another difference is that broadcasting, which causes the main bottleneck in P2P systems, occurs in our system very seldom as the mediator keeps the index of its sources and knows exactly how to split the query and where to route the sub-queries. For this reasons, we do not impose any particular self-organizing network topology on the way cooperating peers are connected. The data channels however, still remain hierarchical. Should the application demand a very high number of cooperating peers, efficient network topologies such as [97] would have to be considered.

Our system offers a number of entry points to the underlying data sources. The user poses a query against one of the mediators and the peers in the system cooperate in order to deliver the answer faster. From this point of view, it resembles Grid computing where several geographically distributed computational resources jointly cooperate on solving a given task.
Similarly to the HMA, the query processing in the CMA starts with the node that receives the initial query. Next, the path index is consulted and the query is split into sub-queries and routed to the appropriate sources. These can be local sources as well as other mediators some of which may have agreed to cooperate on query processing. The results from the “simple” sources are transferred to the initiating mediator—this part of the query plan is identical to that in the HMA. The difference is in treating the cooperating nodes (mediators that implement the communication layer). The mediator with the initial query becomes a coordinator and orchestrates the other cooperating peers. It creates an extended query plan that takes into account the connection speed among different nodes, their processing capacity, as well as available data statistics. For the cooperating nodes, this query plan may consist of a sequence of the following query processing primitives:

- Gather data from the local sources (given a sub-query)
- Receive external data
- Join (a part of) the obtained data
- Ship (a part of) the data to another node
- Provide local statistics (cardinalities, work-load, etc.)

Since the data sources are autonomous, the statistics that are needed to make a good query plan are often not readily available. The mediator gathers these statistics about its subordinate sources in time, with every new query. On top of that, there is a so-called calibration phase during the mediator setup, when several pre-compiled queries are fired: the system tries to assess some of the necessary parameters. All this is however not enough if the data in the underlying sources changes rapidly or if the load of the cooperating peers varies considerably. To tackle these conditions, it is sometimes necessary to intertwine the query processing with query planning. In ideal circumstances, the optimizer makes one plan at the beginning and this plan is executed during the query processing. In more volatile environments or in cases where no or very little statistics are available, the mediator first makes a partial query plan and requests the up-to-date statistics from the cooperating peers, which are subsequently used by the optimizer to improve the initial plan.

As an example, let us consider the case of a heterogeneous cooperative network where two nodes Mediator2 and Mediator3 are connected via a high speed connection while their connection to Mediator1 is considerably slower. Figure 4.7 depicts a message sequence chart of processing a query issued against Mediator1. First, Mediator1 consults its index and splits the query into sub-queries. Then it consults the two cooperative peer mediators and requests statistics about their partial results. This information is then used by the optimizer, which makes a query plan for a two-way semi-join [9] in which Mediator2 and Mediator3 exchange their data (the set of URIs of resources, which is needed to make a

---

1 We assume a heterogeneous network both in terms of network bandwidth, processing power, and (local) data amounts.
join between the two sub-paths). Each of the nodes filters out path instances that will not be present in the result, i.e. they perform a join on their local data with the joining attribute from each other. Subsequently, both peers ship the filtered results to Mediator1 which performs the final join together with the data obtained locally. Due to the semi-join method, all data coming from the cooperative peers will appear in the result, the advantage being that the data transfer on the "expensive" lines (i.e., those that connect Mediator2 and Mediator3 with Mediator1) is kept minimal.

![Diagram](image-url)

Figure 4.7: Query processing and optimization message sequence, cooperative setting.

In the following we focus on data structures and algorithms for querying distributed RDF sources, taking into account the context of our mediator architecture.

### 4.2 Query Processing

This section describes the type of RDF queries we are primarily interested in, and proposes data structures and algorithms for efficient query evaluation.
4.2.1 Data Structures

In order to be able to make use of the optimization mechanisms of the database engines underlying the different repositories, we have to forward entire queries to the repositories. In the case of multiple external models, we can further speed up the process by only pushing down queries to information sources we can expect to contain an answer. The ultimate goal is to push down to a repository exactly that part of a more complex query for which the repository contains an answer. This part can range from a single statement template to the entire query. We can have a situation in which a subset of the query result can be directly extracted from one source, while the remainder has to be extracted and combined from different sources. This situation is illustrated in the following example.

**Example 4.1.** Consider the case in which we want to extract information about research results. This information is scattered across a variety of data sources containing information about publications, projects, patents, etc. In order to access these sources in a uniform way, we use the conceptual model presented in Figure 4.8.

![Conceptual model for the publication portal.](image)

Suppose we now want to ask for the titles of articles by employees of organizations that have projects in the area “RDF”. The path expression of a corresponding SeRQL query would be the following:\(^3\):

```
SELECT A, T
FROM {A} title {T};
author {W} affiliation {O} carriesOut {P} topic "{RDF}"
```

Now let us assume that we have three information sources \(S_1\), \(S_2\), and \(S_3\). \(S_1\) is a publication database that contains information about articles, titles, authors and their af-

---

\(^2\)This example is motivated by the OntoWeb ontology ([http://ontoweb.ontoware.org/Ontology/](http://ontoweb.ontoware.org/Ontology/)).

\(^3\)For the sake of readability we omit namespaces whenever they do not play a role in the discussion.
filiations. $S_2$ is a project database with information about industrial projects, topics, and organizations. Finally, $S_3$ is a research portal that contains all of the above information for academic research.

If we want to answer the query above completely we need all three information sources. By pushing down the entire query to $S_3$ we get results for academic research. In order to also retrieve the information for industrial research, we need to split up the query, push the fragment \{A\} title \{T\}; author \{W\} affiliation \{O\} to $S_1$ and the fragment \{O\} carriesOut \{P\} topic \{"RDF"\} to $S_2$, and join the result based on the identity of the organization.

The example illustrates the need for sophisticated indexing structures for deciding which part of a query to direct to which information source. On the one hand we need to index complex query patterns in order to be able to push down larger queries to a single source. On the other hand we also need to be able to identify sub-queries needed for retrieving partial results from alternate sources.

In order to solve this problem we build upon existing work on indexing complex object models using join indices [95]. The idea of join indices is to create additional database tables that explicitly contain the result of a join over a specific property. At runtime, rather than computing a join the system just accesses the join index relation, which is less computationally expensive. The idea of join indices has been adapted to deal with complex object models. The resulting index structure is a join index hierarchy [126]. The most general element in the hierarchy is an index for elements connected by a certain path $p_{0..n-1}$ of length $n$. Every following level contains all the paths of a particular length from 2 paths of length $n - 1$ at the second level of the hierarchy to $n$ paths of length 1 at the bottom of the hierarchy. In the following, we show how the notion of join index hierarchies can be adapted to deal with the problem of determining information sources that contain results for a particular sub-query.

### 4.2.2 Source Index Hierarchies

The majority of work in the area of object oriented databases is focused on indexing schema-based paths in complex object models. We can make use of this work by relating it to the graph-based interpretation of RDF models. More specifically, every RDF model can be seen as a graph where nodes correspond to resources and edges to properties linking these resources. The result of a query to such a model is a set of subgraphs corresponding to a path expression. A path expression does not necessarily describe a single path, but may have the structure of a tree that can be created by joining a set of paths. Making use of this fact, we first decompose the path expression into a set of expressions describing simple paths. Next, we forward the simpler path expressions to the sources that contain the corresponding information using a path-based index structure, and join the retrieved answers to create the result.
The problem with using path indices to select information sources resides in the fact that the information that makes up a path might be distributed across different information sources (compare Example 4.1). We therefore have to use an index structure that also contains information about sub-paths without losing the advantage of indexing complete paths. An index structure that combines these two characteristics is the join index hierarchy proposed in [126]. Therefore, we take their approach as a basis for defining a source index hierarchy.

**Definition 4.1 (Ontology Path).** Let $G = \langle V, E, L, s, t, l \rangle$ be a labeled graph of an RDF model where $V$ is a set of nodes, $E$ a set of edges, $L$ a set of labels, $s, t : E \rightarrow V$ and $l : E \rightarrow L$.

For every $e \in E$, we have $s(e) = r_1, t(e) = r_2$ and $l(e) = l_e$ if and only if the model contains the triple $(r_1, l_e, r_2)$. A path in $G$ is a list of edges $e_0, \ldots, e_{n-1}$ such that $t(e_i) = s(e_{i+1})$ for all $i = 0, \ldots, n-2$. Let $p = e_0, \ldots, e_{n-1}$ be a path, the corresponding ontology path is the list of labels $l_0, \ldots, l_{n-1}$ such that $l_i = l(e_i)$.

The definition establishes the notion of a path for RDF models. We can now use path-based index structures and adapt them to the task of locating path instances in different RDF models. The basic structure we use for this purpose is an index table of sources that contain instances of a certain path.

**Definition 4.2 (Source Index).** Let $p$ be an ontology path; a source index for $p$ is a set of pairs $(s_k, n_k)$ where $s_k$ is an information source (in particular an RDF model) and the graph of $s_k$ contains exactly $n_k$ paths with ontology path $p$ and $n_k > 0$.

The source index replaces the join index file that is used in [126]. A source index can be used to determine information sources that contain instances of a particular ontology path. If our query contains the path $p$, the corresponding source index provides us with a list of information sources to which we have to forward the query in order to get results. The information about the number of instance paths can be used to estimate communication costs and will be used for join ordering (see Section 5.1.4). So far, the index satisfies the requirement of being able to list complete paths and push down the corresponding queries to external sources. In order to be able to retrieve information that is distributed across different sources, we have to extend the structure based on the idea of a hierarchy of indices for arbitrary sub-paths. The corresponding structure is defined as follows.

**Definition 4.3 (Source Index Hierarchy).** Let $p = l_0, \ldots, l_{n-1}$ be an ontology path. A source index hierarchy for $p$ is an n-tuple $\langle P_n, \ldots, P_1 \rangle$ where

- $P_n$ is a source index for $p$
- $P_i$ is the set of all source indices for sub-paths of $p$ with length $i$ that have at least one entry.
The most suitable way to represent such index structure is a hierarchy, where the source index of the indexed path is the root element. The hierarchy is formed in such a way that the subpart rooted at the source index for a path \( p \) always contains source indices for all the sub-paths of \( p \). This property will later be used in the query answering algorithm. Forming a lattice of source indices, a source index hierarchy contains information about every possible schema sub-path. Therefore we can locate all fragments of paths that might be combined into a query result. At the same time, we can first concentrate on complete path instances and successively investigate smaller fragments using the knowledge about the existence of longer paths. We illustrate this principle in the following example.

**Example 4.2.** Let us consider again the situation in Example 4.1. The ontology path we want to index is given by the list \((\text{author, affiliation, carriesOut, topic})\). The source index hierarchy for this path contains source indices for the following paths:

- \( p_{0,3}:(\text{author, affiliation, carriesOut, topic}) \)
- \( p_{0,2}:(\text{author, affiliation, carriesOut}),
  p_{1,3}:(\text{affiliation, carriesOut, topic}) \)
- \( p_{0,1}:(\text{author, affiliation}),
  p_{1,2}:(\text{affiliation, carriesOut}),
  p_{2,3}:(\text{carriesOut, topic}) \)
- \( p_{0}:(\text{author}),
  p_{1}:(\text{affiliation}),
  p_{2}:(\text{carriesOut}),
  p_{3}:(\text{topic}) \)

Starting from the longest path, we compare our query expression with the index (see Figure 4.9 for an example of index contents). We immediately get the information that \( S_3 \) contains results. Turning to sub-paths, we also find out that \( S_1 \) contains results for the sub-path \((\text{author, affiliation})\) and \( S_2 \) for the sub-path \((\text{carriesOut, topic})\) that we can join in order to compute the final results, because together the two sub-paths make up the path we are looking for.

The source indices also contain information about the fact that \( S_3 \) contains the results for all the sub-paths of our target path. We still have to take this information into account, because in combination with fragments from other sources we might get additional results. However, we do not have to consider joining sub-paths from the same source, because these results are already covered by longer paths. In the example we see that \( S_2 \) will return far less results than \( S_1 \) (because there are less projects than publications). We can use this information to optimize the process of joining results.

A key issue connected with indexing information sources is the trade-off between the required storage space and the computational properties of the index-based query processing. Compared to the index structures used to speed up the query processing within an
information source, a source index is relatively small as it does not encode information about individual data elements in a source. Therefore, the size of the index is independent of the size of the indexed information sources. The relevant parameters in our case are the number of sources \( s \) and the length of the ontology paths \( n \). More specifically, in the worst case a source index hierarchy contains source indices for every sub-path of the indexed ontology path. As the number of all sub-paths of a path is \( \sum_{i=1}^{n} i \), the worst-case\(^4\) space complexity of a source index hierarchy is \( O(s \cdot n^2) \). We conclude that the length of the indexed path is the significant parameter here.

### 4.2.3 Basic Query Answering Algorithm

Using the notion of source index hierarchy we can now define a basic algorithm for answering queries using multiple sources of information. The task of this algorithm is to determine all the possible combinations of sub-paths of the given query path. For each of these combinations, it then has to determine the sources containing results for the path fragments, retrieve these results, and join them into a result for the complete path. The main task is to guarantee that we indeed check all possible combinations of sub-paths for the query path. The easiest way of guaranteeing this is to use a simple tree-recursion algorithm that retrieves results for the complete path, then splits the original path, and joins the results of the recursive calls for the sub-paths. In order to capture all possible splits this has to be done for every possible split point in the original path. The corresponding

---

\(^4\)It is the case where all sources contain results for the complete ontology path.
4.2. QUERY PROCESSING

A semi-formal algorithm is given below (Algorithm 1).

Algorithm 1 Compute Answers.

Require: An ontology path \( p = l_0, \ldots, l_{n-1} \)

Require: A source index hierarchy \( h = (P_n, \ldots, P_1) \) for \( p \)

for all sources \( s_k \) in source index \( P_n \) do
  \( \text{answers} := \) instances of ontology path \( p \) in source \( s_k \)
  \( \text{result} := \text{result} \cup \text{answers} \)
end for

if \( n \geq 2 \) then
  for all \( i = 1 \ldots n - 1 \) do
    \( p_{0..i-1} := l_0, \ldots, l_{i-1} \)
    \( p_{i..n-1} := l_i, \ldots, l_{n-1} \)
    \( h_{0..i-1} := \) Sub-hierarchy of \( h \) rooted at the source index for \( p_{0..i-1} \)
    \( h_{i..n-1} := \) Sub-hierarchy of \( h \) rooted at the source index for \( p_{i..n-1} \)
    \( \text{res}_1 := \text{ComputeAnswers}(p_{0..i-1}, h_{0..i-1}) \)
    \( \text{res}_2 := \text{ComputeAnswers}(p_{i..n-1}, h_{i..n-1}) \)
    \( \text{result} := \text{result} \cup \text{join}(\text{res}_1, \text{res}_2) \)
  end for
end if

return \( \text{result} \)

Note that Algorithm 1 is far from being optimal with respect to runtime performance. The straightforward recursion scheme does not take specific actions to prevent unnecessary work and it does not select an optimal order for joining sub-paths either. We can improve this situation by using knowledge about the information in the different sources and by performing query optimization.

4.2.4 From a Single Path to Ad-hoc Queries

The index structure of the previous section provides query optimization support for individual queries which are a sub-path of the ontology path of the index. One particular source index hierarchy covers one particular path in the RDF model. This kind of structure is quite suitable for frequently posed queries, and it is worth while to maintain these structures.

In order to cover optimization support for arbitrary queries one can think along two lines. One can construct source index hierarchies for a set of simple paths covering the entire RDF model graph. This approach is appropriate for the situation in which there the RDF model is not too extensive. Only a simple auxiliary structure is needed to map a given query to a hierarchy in which it is a sub-path.
Alternatively, one can construct a more general structure from which the source hierarchy covering the optimization information for the query at hand can be obtained. This approach has the advantage that the information about the common sub-paths is stored only once and is easier to maintain. In particular, the information about short (length 1 or 2) paths will be replicated many times in case of the other alternative.

**Example 4.3.** Consider the case where we also want to extract information about potential collaborators from organizations already involved in a project. Again we use the OntoWeb research ontology, shown in part in Figure 4.8.

Suppose we now want to ask for the names of employees of organizations that are involved in carrying out projects in which Mr. Houben is involved as a member. The corresponding SeRQL query would be the following:

```sql
SELECT N
FROM {E} name {N};
affiliation {O} carriesOut {P} member {} name {"Houben"}
```

We see that this SeRQL query overlaps with the one in example 4.1: it shares the sub-path affiliation-carriesOut and all sub-paths thereof.

There are several options for such a generalized structure. The basic idea is to create an index based on the path-expressions that points to the allocation information about the data sources. Interesting options are the Index Fabric [27] and the B+ tree [69]. The former uses a multi-layered tree structure where the leaf nodes point to the source data. A compression scheme is used to deal with lengthy index key expressions. The particular compression scheme introduced in [27] is optimized for storing XML data and needs to be adapted for storing source index hierarchies.

The B+ tree index can be set up in the same way, using the path expressions as index keys and the leaf nodes to point to the source indices corresponding to the path expression value of the key. The B+ tree has the advantage of being sorted. This implies that all the left children in the source index hierarchy that have the same key prefix will be easily retrieved at the leaf level. A requirement is that the node labels have to be concatenated in such a way that for key combinations of which one is the prefix of the other, the longer one precedes the shorter one in the sorting order, e.g. "author-affiliation" < "author". An example of such a tree that incorporates both example paths is depicted in Figure 4.10. In this example the tree can accommodate 3 key-values and 4 pointers per node. We have abbreviated the labels of the edges, i.e. affiliation becomes AF, author becomes AU, and so on. Algorithm 2 presents a simple procedure to extract a source index hierarchy from the B+-tree.
4.3. RELATED WORK

In this chapter we focused mainly on architectures for distributed RDF querying and on optimization techniques such as indexing and join ordering. Relevant related work is described in the remainder of this section.

4.3.1 Distributed Architectures

The centralized mediator architecture is just one of a number of architectures for the access to the contributing resources. On the other end of the integration spectrum are the highly
decentralized P2P systems which by design offer a high degree of interoperability and robustness. However, the description of the resources in the early P2P systems was rather limited, and consequently also their support for querying. In fact, the earliest P2P systems, such as Napster\(^5\), were (audio)file-sharing systems that provided access through the specification of a few keywords, such as a title or an artist. These systems are characterized by a high, but varying number of peers (nodes in the network) and a high redundancy of the files to be shared.

More advanced querying is supported in the schema-based P2P systems such as the Edutella system [21], that provides an RDF-based infrastructure. Here queries can be posed against locally available schemata and are decomposed and distributed to peers that are known (on the basis of local routing indices) to be able to contribute their information resources. Edutella also tackles the problem of dynamic network evolution by proposing an optimal network topology with respect to the message broadcast mechanism, and algorithms for maintaining the topology [97]. The topology maintenance is crucial for highly volatile P2P networks where sources often join and leave the federation. However, this is less of an issue for the more settled Hera integration context where sources are carefully selected and mapped to our conceptual model.

Schema-aware querying is also offered in the Piazza system [51]. Here a bottom-up procedure for distributing queries is used, based on the locally available mappings of the schema of a peer to the schemata of its neighbors. So far, the focus in improving the query performance in schema-aware P2P systems has been on the reduction of the network traffic by making sure that the queries are propagated to only those peers that can be expected to provide an answer. Because of the highly dynamical character of these systems not much emphasis has been placed yet on query optimization.

### 4.3.2 Index Structures for Object Models

There exists a large body of research on indexing object oriented databases. The aim of this work was to speed up querying and navigation in large object databases. The underlying idea of many existing approaches is to regard an object database as a directed graph, where objects correspond to nodes and object properties to links [103]. This view directly corresponds to RDF data, that is also often regarded as a directed graph. Indices over such graph structures now describe paths in the graph based on a certain pattern normally provided by the schema. Different indexing techniques vary on the kind of path patterns they describe and on the structure of the index. Simple index structures only refer to a single property and organize objects according to the value of that property. Nested indices and path indices cover a complete path in the model that might contain a number of objects and properties [10]. In RDF as well as in object oriented databases, the inheritance relation plays a special role as it is connected with a predefined semantics. Spe-\(^5\)http://www.napster.com
cial index structures have been developed to speed up queries about such hierarchies and have recently been rediscovered for indexing RDF data [26]. In the area of object-oriented database systems, these two kinds of indexing structures have been combined resulting in the so-called nested inheritance indices [11] and generalized nested inheritance indices [103]. These index structures directly represent implications of inheritance reasoning, an approach that is equivalent to indexing the deductive closure of the model.

### 4.4 Summary of Distributed RDF Query Processing

In this chapter we focused on mediating architectures and distributed RDF query processing, addressing research question 3.

In order to guarantee the freshness of the retrieved data, the Hera integration framework adopts a mediating architecture which implements the on-demand retrieval paradigm in which data is collected on-the-fly from the sources during the query evaluation process. In this chapter we presented several types of mediating architectures that follow this principle. The basic mediator architecture establishes transparent access to the underlying repositories containing the RDF data instances that are to be retrieved for the conceptual model. The hierarchical mediator architecture extends the basic architecture in the sense that the mediated sources can be mediators themselves and in turn hide a set of distributed sources. The cooperative mediator architecture moves us closer towards the P2P paradigm; in this architecture (some) mediators offer processing capabilities, especially the join evaluation, to other nodes in the mediating network.

One of the main issues in distributed processing of RDF queries is to distribute the sub-paths of a particular query to the appropriate sources. We proposed the path index data structure which allows for efficient access to the mapping information form the integration model. Using this data structure, we devised an algorithm for the efficient distribution of the RDF path queries. The main idea is to use the path index to maximize the sub-queries that are being pushed to the underlying repositories. This both enables the local (instance based) optimization, employs parallelism (the underlying sources work in parallel), and reduces the data traffic by pruning extraneous instances.
Chapter 5

Relational Optimization Techniques for RDF Querying

This chapter focuses on details of the distributed RDF query optimization techniques. In order to optimize the query processing in our distributed framework, in particular the phase where the mediator assembles the partial results, we map the evaluation of the RDF path queries onto the join ordering problem from the database research. This allows us to apply some of the optimization techniques devised by researchers in the database field. We explain the testing environment and provide the performance evaluation of our optimizer. We first devise an optimization solution for linear path queries, later we generalize it to arbitrarily shaped join queries. The chapter also describes the implementation of our mediator component which uses the ontology mapping information introduced in Chapter 3, and acts as our integration engine. The mediator can be embedded in an existing RDF storage and querying engine—Sesame [18], acting for the outside world as its distributed embodiment.

This chapter is structured as follows. In Section 5.1 we relate the problem of RDF path querying to that of join ordering in the field of relational databases. We introduce a cost model for processing queries in the distributed architecture and show its use in optimizing query execution, as a basis for the two-phase optimization heuristics for join ordering. Section 5.2 describes the implementation of our mediator which can be deployed either as a standalone application or it can extend the Sesame system acting as its distributed implementation. At the end of this section we describe our test environment and provide the performance evaluation. Section 5.3 reviews the related work on query optimization, particularly focusing on the join ordering problem. Section 5.4 concludes this chapter with a short summary.
5.1 Query Optimization

In the previous chapter we described a light-weight index structure for distributed RDF querying. Its main task is to index ontology paths with respect to the underlying sources that contain them. Compared to instance-level indexing, our approach does not require creating and maintaining oversized indices since there are far fewer sources than there are instances. Instance indexing would not scale in the web environment, and as mentioned above in many cases it would not even be applicable, e.g. when sources do not allow replication of their data (which is what instance indices essentially do). The downside of our approach is that query answering without the index support at the instance level is much more computationally intensive. Moreover in the Hera context, the user queries are often extended and frequently exceed the size\(^1\) that can be computed easily by using brute force. Therefore, in this section we focus on query optimization as an important part of a distributed RDF query system. We try to avoid re-inventing the wheel and once again seek for inspiration in the database field, making it applicable by “relationalizing” the RDF model.

Each single ontology path \(p_i\) of length 1 (also called 1-path) can be perceived as a relation with two attributes: the source vertex \(s(p_i)\) and the target vertex \(t(p_i)\). An ontology path of length more than 1 is modeled as a set of relations joined together by the identity of the adjacent vertices, essentially representing a chain query of joins as defined in Definition 5.3. This relational view over an RDF graph offers the possibility to re-use the extensive research on join optimization in databases [9; 55; 57; 106; 113].

Taking into account the (distributed) RDF context of the join ordering problem there are several specifics to note when devising a good query plan. As in distributed databases, communication costs significantly contribute to the overall cost of a query plan. Since in our case the distribution is assumed to be realized via an IP network with a variable bandwidth, the communications costs are likely to contribute substantially to the overall processing costs, which makes the minimization of the data transmission across the network very important.

**Distribution Aspects**

The distribution of data often follows certain patterns or dependencies. Some of them are imposed by the geographical nature of distribution, some are introduced due to the diversity of source domains. For instance, one source may focus on a domain \(A\) whereas another source may focus on a domain \(B\). If there is no overlap between the instances of the two domains, we can conclude that there is no overlap between the ontology paths covered by those sources either. Such information, when available, can be used by the optimizer and the path expressions from the two sources can be evaluated independently from each other, using a set union operation at the end to produce the final result. This

\(^1\)Especially, the length of the path expression.
effectively reduces the number of tuples which are to be joined and thus also the overall
costs. In the following we provide formal definitions of the types of data fragmentation
that may occur in our distributed context.

**Definition 5.1 (Horizontal fragmentation).** Let \( p \) be an ontology path and let \( X(p) \)
denote those instances of \( p \) residing on the source \( X \). We say that \( p \) is (a) horizontally
fragmented, or (b) partially horizontally fragmented, if there exist at least two sources \( X \)
and \( Y \) such that \( |X(p)| > 0, |Y(p)| > 0 \) and (a) \( X(p) \cap Y(p) = \emptyset \), or (b) \( X(p) \neq Y(p) \).

**Example 5.1.** The ontology path (author, affiliation) is horizontally fragmented because
there exist two different sources \((S_1, S_3)\) that contain instances for this ontology path.

**Definition 5.2 (Vertical fragmentation).** Let \( p \) be an ontology path and let \( r \) and \( s \)
denote two not overlapping sub-paths of \( p \). Let \( Ext(r) = \cup \{X_i(p), |X_i(p)| \geq 0 \} \) and
similarly \( Ext(s) = \cup \{X_j(p), |X_j(p)| \geq 0 \} \), and let \( I_X, J_X \) denote the two sets of sources
satisfying the above conditions for \( r \) and \( s \) respectively. Then \( p \) is (a) vertically fragmented,
or (b) partially vertically fragmented on \( r \) and \( s \) if (a) \( I_X \cap J_X = \emptyset \), or (b) \( I_X \neq J_X \).
Informally, we say that \( p \) is (partially) vertically fragmented if (some) instances of \( r \) reside
on different sources than (some of) the instances of \( s \).

**Example 5.2.** The ontology path (author, affiliation, carriesOut, topic) is partially verti-
cally fragmented since its two subpaths (author, affiliation) and (carriesOut, topic) reside
on two different sets of sources, i.e. \( \{S_1, S_3\} \) and \( \{S_2, S_3\} \), respectively.  

Note that no path of length 1 is vertically fragmented. This results directly from
Definition 5.2 and from the atomicity of RDF triplets. The horizontal fragmentation for a
1-path relation can still occur, i.e. several sources can contain instances of the same 1-path.

The statistics about the data and about the source performance are important for the
optimizer in order to make the right decision. In general, the more the optimizer knows
about the underlying sources and data, the better optimized the query plan is. However,
taking into account the autonomy of the sources, the necessary statistics do not have to
be always available. We design our mediator to cope with incomplete statistical informa-
tion in such a way that the missing parameters are estimated as being worse than those
that are known (pessimistic approach). Naturally, the performance of the optimizer is
then lower but it increases steadily when the estimations are made more realistic based
on the actual response from the underlying sources; this is also known as optimizer cali-
bration. As indicated above, the computational capabilities of the underlying sources may
vary considerably. We distinguish between sources that can only retrieve the selected local
data (pull up strategy) and sources that can perform joins of their local and incoming
external data (push down strategy), thus offering computational services that could be
used to achieve both a higher degree of parallelism and lower data transmission over the
network, e.g., by applying semi-join reductions [9] as described in Section 4.1.3, Figure 4.7.

---

2The fragmentation is considered only partial as the two sets have a common element \( (S_3) \).
5.1.1 Relational Approach for Processing RDF Path Queries

As mentioned earlier, we can perceive an RDF model as a set of relations on which we can apply optimization results from the area of relational databases. In this context, the problem of join ordering arises, when we want to compute the results for ontology paths from partial results obtained from different sources. Processing a (linear) path query corresponds to the problem of computing the result of a chain query as defined below:

**Definition 5.3 (Chain Query).** Let \( p \) be an ontology path composed from the 1-paths \( p_1, \ldots, p_n \). The chain query of \( p \) is the \((n-1)\)-join \( p_1 \bowtie_{q(p_1)=s(p_2)} p_2 \bowtie_{q(p_2)=s(p_3)} p_3 \bowtie \cdots p_n \), where \( s(p_i) \) and \( t(p_i) \) are returning the identity of a source and of a target node, respectively. As the join condition and attributes follow the same pattern for all joins in the chain query, we omit them whenever they are clear from the context.

In other words, to follow a path \( p \) of length 2 means performing a join between the two paths of length 1 that \( p \) is composed of. When a source is capable of answering a longer path (this information is obtained from the index hierarchy), we leave the computation of the inner joins at the source and consider only the beginning and the end of that path\(^3\), which are to be joined with data coming from different sources.

The problem of join optimization is to determine the order in which the joins should be computed, such that the overall response time for computing the path instances is minimized.\(^4\) Note that a chain query in Definition 5.3 does not include explicit joins, i.e. those specified in the *Where* clause or by assigning the same variable names along the path expression. When we append these explicit joins, the shape of the query usually changes from a linear chain to a query graph containing a circle or a star, making the join ordering problem NP-hard [96]. We discuss the creation of query plans from explicit SeRQL joins in Section 5.1.6.

5.1.2 Solution Space

Disregarding the solutions obtained by the commutativity of joins, each query execution plan can be associated with a sequence of numbers that represents the order in which the relations are joined. We refer to this sequence as footprint of the execution plan.

**Example 5.3.** Let us consider the same model as introduced in Example 4.1. In order to demonstrate the join ordering problem we assume here a total distribution, i.e. the sources contain only 1-paths and all processing is done at the mediator. For brevity reasons we introduce the following name substitutions: the concept names *Article, Employee, Organization, Project, ResearchTopic* become \( a, b, c, d, e \), respectively; the property names *author, affiliation, carriesOut, topic* are substituted with \( 1, 2, 3, 4 \), respectively. Figure 5.1 presents two possible execution plans and their footprints.

---

\(^3\)This essentially transforms it into a 1-path.

\(^4\)In case the sources offer also join capabilities the problem is not only in which order but also where the joins should take place.
5.1. QUERY OPTIMIZATION

The solution space consists of the query plans (their footprints), which can be generated by considering the order of join operators and the join operands (the sub-paths delivered by the sources). When we allow for an arbitrary order of joins the resulting query plans can be represented as so-called bushy trees where the operands of a join can be both a base relation and a result of a previous join.

A special case of a general execution plan is the so-called right-deep tree which has the left-hand join operands consisting only of base relations. In this specially shaped query tree an execution pipeline of length $n - 1$ exists that allows both for easier parallelization and for decreasing the response time [55]. As mentioned earlier, this property is very useful in the context of WIS where the system needs to generate the first pages of the presentation as soon as possible, while the rest can be completed while the user is browsing the first lot.

Figure 5.2 depicts possible query execution plans for a path expression of length 4. The solid and the dashed gray lines indicate the execution plans of footprints $(3,2,4,1)$ and $(2,1,4,3)$, respectively. Note that in the case of bushy trees there might be several footprints associated with one query tree. For instance, the bushy tree in Example 5.3 can be evaluated in a different order yielding two more footprints: $(2, 4, 1, 3)$ and $(4, 2, 1, 3)$. In our current approach, these footprints would be equivalent with respect to the cost they represent. However, treating them independently allows us to consider also the semi-join

---

A base relation is that part of the path which can be retrieved directly from one source.

<table>
<thead>
<tr>
<th>Path expression</th>
<th>a→b→c→d→e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Footprints</td>
<td>(3,2,4,1)</td>
</tr>
<tr>
<td>Query trees</td>
<td>abcde</td>
</tr>
<tr>
<td></td>
<td>a ⊕ bcde</td>
</tr>
<tr>
<td></td>
<td>e ⊕ bcde</td>
</tr>
<tr>
<td></td>
<td>b ⊕ cd</td>
</tr>
<tr>
<td></td>
<td>c ⊕ d</td>
</tr>
</tbody>
</table>

Right-deep tree | Bushy tree

Figure 5.1: Two possible query executions and their footprints.
optimization [9] where their cost might differ considerably. If also the order of the join operands matters, i.e. the commutativity law is considered, the sequence of the operands of each join is recorded in the footprint as well. Treating these cases independently also means that our solution space is larger than that which considers join ordering as parenthesizing expressions [87].

If the commutativity of join is taken into account, there are \( \left( \frac{2^n}{n} \right) n! \) different possibilities of ordering joins and their individual constituents [106]. However, in case of memory-resident processing (as adopted by our mediator), the possibilities generated by the commutativity law can be neglected for some of the join methods as these possibilities mainly play a role in the cost model minimizing disk-memory operations. We discuss this issue further in Section 5.1.3.

### 5.1.3 Cost Model

The main goal of query optimization is to reduce the computational cost of processing the query both in terms of the transmission cost and of the cost of performing join operations on the retrieved result fragments. In order to determine a good strategy for processing a query, we have to be able to exactly determine the cost of a query execution plan and to compare it to costs of alternative plans. For this purpose, we capture the computational costs of alternative query plans in a cost model that provides the basis for the optimization
As mentioned earlier, we adopt the memory-resident paradigm, and the cost minimization we are trying to perform is equivalent to the minimization of the total execution time. There are two main factors that influence the resulting cost in our model. First is the cost of data transmission to the mediator, and second is the cost of data processing.

**Definition 5.4 (Transmission Cost).** The transmission cost of path instances of the ontology path $p$ from a source $X$ to the mediator is modeled as $TC_p = C_{init_X} + \mid p\mid \times L_{h} \times \parallel URI_X \parallel \times C_X$ where $C_{init_X}$ represents the cost of initiating the data transmission, $\mid p\mid$ denotes the cardinality, $L_{h}$ stands for the length of the ontology path $p$, $\parallel URI_X \parallel$ is the size of a URI at the source $X$\(^6\) and $C_X$ represents the transmission cost per data unit from $X$ to the mediator.

Since we apply all reduction operations (e.g. selections and projections) prior to the data transmission phase, the data processing mainly consists of join costs. The cost of a join operation is influenced by the cardinality of the two operands and the join method which is utilized. As we already pointed out, there are no instance indices at the mediator side that would allow us to use some join “shortcuts”. In the following we consider two join methods: a nested loop join and a hash join, both without additional indexing support.

**Definition 5.5 (Nested loop join cost).** The processing cost of a nested loop join of two relations $p$, $r$ is defined as $NJC_{p,r} = \mid p\mid \times \mid r\mid \times K(p, r)$, where $\mid x\mid$ denotes the cardinality of the relation $x$ and $K(p, r)$ represents the cost of the identity comparison.

Note that the nested loop join allows for a more sophisticated definition of object equality than a common URI comparison. In particular, if necessary, the basic URI comparison can be complemented by (recursive) comparisons of property values or mapping look-ups. This offers room to address the issue of URI diversity, also known as the designation problem, when two different URIs refer to the same real-life object.

**Definition 5.6 (Hash join cost).** The processing cost of a hash join of two relations $p$, $r$ is defined as $HJC_{p,r} = I \times \mid p\mid + R \times \mid r\mid \times B$, where $\mid x\mid$ denotes the cardinality of the relation $x$, $I$ represents the cost of inserting a path instance in the hash table (the building factor), $R$ models the cost of retrieving a bucket from the hash table, and $B$ stands for the average number of path instances in the bucket.

Unlike the previous join method, the hash join algorithm assumes that the object equality can be determined by a simple URI comparison, in other words that the URI references are consistent across the sources. Another difference is that in the case of the nested loop join for in-memory relations the join commutativity can be neglected, as the query plan produced from another query plan by the commutativity law will have exactly the same cost. However, in the case of the hash join method the order of operands

---

\(^6\)Different sources may model URIs differently, however, we assume that at the mediator all URIs are represented in the same way.
influences the cost and thus the solution space must also include those solutions produced by the commutativity law.

**Definition 5.7 (Query plan cost).** The overall cost of a query plan $\theta$ consists of the sum of all communication costs and all join processing costs of the query tree. $\text{QPC}_\theta = \sum_{i=1}^{n} T\text{C}_p + \text{PC}_\theta$, where $\text{PC}_\theta$ represents the join processing cost of the query tree $\theta$ and it is computed as a sum of recurrent applications of the formula in Definition 5.5 or 5.6 depending on which join method is utilized. To compute the cardinality of non-base join arguments, the join selectivity is used. The join selectivity $\sigma$ is defined as the ratio between the tuples retained by the join and those created by the Cartesian product: $\sigma = \frac{|p \Join r|}{|p \times r|}$.

As it is not possible to determine the precise join selectivity before the query is evaluated, $\sigma$ for each sub-path join is assumed to be estimated and available in the source index hierarchy. After the evaluation of each query, the initial $\sigma$ estimates are improved and made more realistic.

### 5.1.4 Heuristics for join ordering

While the join ordering problem in the context of a linear/chain query can be solved in a polynomial time [91], we have to take into account the more complex problem when also the explicit joins are involved which is proved to be NP-hard [96]. It is apparent that evaluating all possible join strategies for achieving the global optimum becomes quickly unfeasible for a larger $n$. In these cases we have to rely on heuristics that compute a “good-enough” solution given the constraints. In fact, this is a common approach for optimizers in interactive systems. There, optimization is often about avoiding bad query plans in very short time, rather than devoting a lot of the precious CPU time to find the optimal plan, especially when it is not so uncommon that the optimal plan improves the heuristically obtained solutions only marginally.

Heuristics for the join ordering problem have been studied extensively in the database community. In this work we adopt the results of comparing different join ordering heuristics from [106]. Inspired by this survey, we choose to apply the two-phase optimization consisting of the iterative improvement (II) algorithm followed by the simulated annealing (SA) algorithm [113]. This combination performs very well on the class of queries we are interested in, both in the bushy and the right-deep tree solution space, and degrades gracefully under time constraints.

The II algorithm is a simple greedy heuristics which accepts any improvement on the cost function. II randomly generates several initial solutions, taking them as starting points for a walk in the chosen solution space. The actual traversal is performed by applying a series of random moves from a predefined set. The cost function is evaluated for every such move, remembering the best solution so far. The main idea of this phase is to descent rapidly into several local minima assuring the aforementioned graceful degradation. For
5.1. QUERY OPTIMIZATION

each of the sub-optimal solutions, the second phase of the SA algorithm is applied. The task of the SA phase is to explore the “neighborhood” of a candidate solution more thoroughly, hopefully lowering the cost.

Algorithm 3 Simulated annealing algorithm

Require: start solution \( s_{Solution} \)

Require: start temperature \( s_{Temp} \)

\[
\text{solution} := s_{Solution} \\
\text{bestSolution} := \text{solution} \\
\text{temp} := s_{Temp} \\
\text{cost} := \text{Cost(bestSolution)} \\
\text{minCost} := \text{cost}
\]

repeat

repeat

\[
\text{newSolution} := \text{NEW(solution)} \\
\text{newCost} := \text{Cost(newSolution)}
\]

if \( \text{newCost} \leq \text{cost} \) then

\[
\text{solution} := \text{newSolution} \\
\text{cost} := \text{newCost}
\]

else if \( e^{-\frac{\text{newCost} - \text{cost}}{\text{temp}}} \geq \text{RAND}(0..1) \) then

\[
\text{solution} := \text{newSolution} \\
\text{cost} := \text{newCost}
\]

end if

if \( \text{cost} < \text{minCost} \) then

\[
\text{bestSolution} := \text{solution} \\
\text{minCost} := \text{cost}
\]

end if

until equilibrium reached

DECREASE(\text{temp})

until frozen

return \text{bestSolution}

The pseudo-code of the SA phase is presented in Algorithm 3. It takes a starting point/solution from the II phase, and similarly to II performs random moves from a predefined set accepting all cost improvements. However, unlike II, the SA algorithm can accept with a certain probability also those moves that result in a solution with a higher cost than the current best solution. The probability of such acceptance depends on the temperature of the system and the cost difference. The idea is that at the beginning the system is hot and accepts easier the moves yielding even solutions with higher costs. However, as the temperature decreases the system is becoming more stable, strongly preferring those solutions with lower costs. The SA algorithm improves the II heuristics by making the stop condition less prone to get trapped in a local minimum. SA stops when the temperature
drops below a certain threshold or if the best solution so far was not improved in a number of consecutive temperature decrements—the system is considered frozen. There are two sets of moves: one for the bushy solution space and one for the right-deep solution space. For details we refer the reader to [113].

Figure 5.3: Acceptance probability with respect to the temperature and the cost difference.

Figure 5.3 shows the acceptance probability dependency in the SA phase, computed for the range of parameters that we used in our experiments.

5.1.5 Making the Query Plan Robust

The cooperative mediator architecture introduced earlier has the potential to bring benefits both in terms of improved performance and flexibility. However, since the cooperating nodes are still autonomous, their agreement to cooperate does not guarantee exclusivity, i.e. they can accept requests from several nodes at the same time. Therefore, it is desirable that the query optimization in such a distributed setting is resilient (at least to some extent) to sudden changes of workload (and therefore performance) which may occur in some of the cooperating nodes.

Intertwining query planning with query processing is the first step to make the query answering more robust. By splitting the query plan into smaller pieces (which may change
5.1. QUERY OPTIMIZATION

during the course of answering the query once the necessary statistics are present) the optimizer reacts better on the actual situation of the entire system. Every such split however introduces the overhead of statistics retrieval and of re-optimizing the rest of the plan. Thus, it is important to keep the number of planing preemptions minimal. The chosen (sub)plan should not turn from a reasonably good plan to a very bad plan if one or more sources do not perform as expected. There is an inherent tradeoff between how robust or safe the plan can be and how good it is, given that the initial assumptions hold.

The translation of the query optimization problem into a search problem led us to identify the following main dimensions: the join ordering problem and the problem of distributing the work among the cooperating nodes. As described earlier, our optimizer implements a heuristic algorithm that walks the problem space evaluating the cost function (the cost model). The idea of finding a robust query plan corresponds in hill-climbing terminology to finding a high plateau. In such a plan, even if some of the parameters change the overall cost will not deteriorate much. On the other hand, even a very good solution that is found on the border of an abyss in our search space should be avoided since a small change in the underlying assumptions can cause a big cost penalty. In order to obtain a relatively stable solution the optimizer explores the neighborhood of \( k \) top plans and picks the one which meets the “safety” requirements and offers a good performance tradeoff.

5.1.6 From Chain Queries to General Join Queries

So far we were only concerned with chain queries as defined in Definition 5.3. In this section we extend our approach to general join queries. As already pointed out, a chain query does not include explicit joins which occur in SeRQL queries when (some) path expressions share the variable name or some of the variables are explicitly defined to be equal in the Where clause. When we do consider these explicit joins, the path query can take an arbitrary shape e.g. circle or star. This section generalizes our approach to these kinds of queries, extending our optimization to arbitrary SeRQL path expressions \(^7\). To explain this extension, let us consider the following example.

Example 5.4. Figure 5.4 introduces a new conceptual model for accessing the distributed information about authors and their publications. This model contains four concepts which are interconnected with five properties. In this example, we assume a complete vertical fragmentation (see Definition 5.2), i.e. every property is gathered from a different source.

The query under investigation involves all concepts and properties of our schema:

“Retrieve all authors and their papers, where the author is an expert in the topic that concerns the paper and he is also mentioned on the front-page of the paper.”

A SeRQL translation of this query is presented below. Note that some path expressions

\(^7\)Note that this still does not cover the SeRQL language in its entirety, e.g. the construct clause is not supported. The BNF grammar of the subset of the SeRQL language that we support at the moment is presented in Appendix B.
share the names of the variables and the Where clause declares two different variables (TA and TP) to be equal. This is to illustrate the two ways in which explicit joins can be introduced in SeRQL queries. These explicit joins make a strong semantic impact on the retrieved data. For instance, if we omit the Where clause, we would retrieve also those papers which were written by authors that are not experts in the topic of the paper.

As mentioned above, the iterative improvement heuristic starts with a number of randomly generated (join) query plans. However, a transition from a general SeRQL query to a query plan is more complex than when only chain queries are considered. In the case of a chain query, the join condition always includes only two connecting attributes at a time. In a general case, the join condition can contain several conjuncts depending on the shape of the query.

In order to obtain a starting query plan from a general SeRQL query we proceed as follows.

- We first normalize the SeRQL query in the sense that all variables that are declared explicitly equal in the Where clause are given identical names (variables TP and TA are both renamed to T). Such normalized query can be represented by a query graph (see Figure 5.5, top), where nodes correspond to the query variables and edges stand for the properties declared in the From path expressions.
• We apply again the relational view of RDF properties and perceive every edge in the query graph as a relation with two attributes represented by the variable names. For instance, the edge $\text{authorOf}$ from the query graph represents a relation $\text{authorOf}$ with two attributes $A$ and $P$.

• To create a join graph (see Figure 5.5, bottom) we represent these relations as nodes and connect every two nodes that share an attribute. As depicted in Figure 5.5, every edge in the join graph can be interpreted as a join condition between the two connected relations.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{query_graph}
\caption{Query graph (top), join graph (bottom).}
\end{figure}

\footnote{If the nodes share more attributes, we create one edge for each such attribute.}
Next, we randomly order the edges of the join graph (the order is depicted by numbers above the edges) and recursively fold the join graph by combing the two nodes that are connected by the edge with the smallest number. The folding sequence for our join graph is depicted in Figure 5.6. Dashed lines indicate the edges which are removed in that particular folding step.\(^9\)

The entire folding sequence essentially represents a starting query plan: a folding step is equivalent to a join operator and every edge which is removed in that particular step represents one conjunct in the join condition (see Figure 5.7).

\(^9\)For the sake of clarity, the node names were abbreviated to their starting letters.
In this section we describe the implementation of our mediator prototype and its merger with the Sesame engine. We also provide the performance evaluation of the aforementioned query optimization techniques.

Figure 5.8 shows the class association diagram of the Hera mediator. The path index is represented by the class PropertyIndex. Except of the usual put() and get() methods, this class also offers methods for handling articulations of our integration model (addArticulations(), mergeArticulations()).

The actual query processing is implemented by a number of collaborating classes. The CMQuery class is responsible for parsing a SeRQL query. As mentioned earlier, we focused on RDF path queries, which are translated into Select-From-Where SeRQL queries. We remark that the SeRQL language \cite{20,19} offers other features such as the Construct option which at the moment are not supported by our prototype. The BNF grammar of the SeRQL subset that our prototype implements is described in Appendix B.

The Distributer class is responsible for splitting the user query among the underlying sources based on the path index. Once the sub-queries and the sources to which they are supposed to be routed are known, the PETranslator class translates the CM path expressions to those of the sources. The last step before the sub-queries are shipped to the sources is the extension to retrieve the proper idBy information and the inclusion of the concerning sub-conditions from the initial Where clause.

The shipping of the queries and the actual data retrieval are executed in a multi-threaded fashion to achieve parallel query evaluation at the sources. The QueryResultTableConstructor class orchestrates the result assembly process. It instantiates a QueryResultTableJoiner class for retrieval and joining of the result tables (three different join variants are implemented: nested join, hash join, sort-merge join). The SAJoinOptimizer class takes care of the join ordering optimizations as described in Section 5.1.4. After every query execution the mediator updates the data statistics; these are used in the next query evaluation.
to improve the optimization process. For further details concerning the implementation of the Hera mediator we refer to [6].

Figure 5.8: UML class association diagram of the Hera mediator [6].
In the following we briefly describe the merger of the Hera mediator with the Sesame engine in order to provide distributed query facilities.

5.2.1 The Merger of the Hera Mediator and Sesame

From the user perspective, both the Hera mediator and the Sesame engine [18] act pretty much the same: they both accept SeRQL queries and deliver the data that satisfies them. The merger of the two systems, in which we use the Sesame front-end and the Hera mediator as the back-end offers several advantages from both Hera’s and Sesame’s point of view. First, our mediator benefits from the elaborated communication layer of Sesame that features different access protocols including SOAP and HTTP. Second, the Sesame engine offers several RDF query language front-ends, including RQL[63], RDQL [101], and SeRQL [19]. Embedding the Hera Mediator into Sesame could enable the use of additional query languages which were initially not supported. In exchange, the Sesame engine becomes distributed and gains the integration facilities of the Hera mediator which allows the querying of multiple semantically heterogeneous repositories.

![Layered Architecture of the Sesame System.](image)

The layered architecture of the Sesame framework shown in figure 5.9 facilitates a flexible extension of the Sesame system. The core of this architecture is the SAIL API
which provides a common way of accessing RDF information independent of the storage model and of the query language front-end. The API defines a number of interfaces for interacting with RDF models that have been implemented for different storage systems (e.g. JDBC relational database, native storage, or in-memory storage).

By implementing the required SAIL interfaces the Hera mediator essentially fits in the Sesame framework as an alternative storage system, as depicted in Figure 5.9 (bottom), virtually hiding the distribution of the underlying sources.

In the next section we describe the performance evaluation of our prototype, focusing in particular on the optimization techniques that were described earlier.

\[\text{Figure 5.10: The distributed testing environment.}\]

### 5.2.2 Performance Evaluation

We first tested our integration framework on the test cases available within the Hera project, including for instance a virtual museum application and a comics portal.\(^{10}\) For these applications the mediator was able to answer queries within milliseconds, which was more than sufficient as the user was provided with almost immediate response. To find the bottlenecks of our integration framework we needed to use larger data sets. For this purpose, we synthesized an RDFS schema of approximately 50MB and instantiated it with 500MB of RDF instances. Note that a schema-instance ratio of 10% is quite large; in normal circumstances, the size of the schema seldom reaches even 1% of the size of the

\(^{10}\)http://wwwis.win.tue.nl:8090/Hera2WebApp/
instances. This represented for us a worst case scenario since the mediator has to join partial path results—the bigger the schema, the more potential paths to join.

Our generated data set was distributed among several computers, as depicted in Figure 5.10. Three repositories and the mediator server were situated at the local network, the fourth repository was geographically distributed (located at Vrije Universiteit Amsterdam). The underlying Sesame repositories used the MySQL back-end (A) and the in-memory repository (B,C,D) to store the RDF instances. The generated integration model used the \textit{idByValue} mechanism to determine the instance identity.

The first set of experiments concerned the improvement gained by path indexing. We compared the performance of query processing with a complete path index with that with a 1-path index. The results of this comparison are depicted in Figure 5.11. As we expected, the complete path index considerably outperforms the 1-path index. The reason for this is that the use of a complete index allows for pushing large sub-queries to the underlying sources. This effectively removes the joining burden from the mediator, reduces the data transmission, and increases the parallelism. Opposed to that, the 1-path index always breaks queries into single paths; after the retrieval, all these 1-paths must be joined by the mediator.
While the path indexing is clearly beneficial especially for larger results sets, the joining at the mediator still represents a bottleneck. In order to minimize the joining time, which due to different cardinalities and join selectivities largely depends on the order in which the

Figure 5.12: The SA optimization improvement, uncalibrated system.

Figure 5.13: The SA optimization improvement, calibrated system.
5.3. RELATED WORK

joins are performed, we deployed our iterative improvement heuristic combined with the simulated annealing. To test the impact of these ordering heuristics, we considered a path query consisting of 15 individual properties. We compared the joining times of the initial query plans with those produced by the join ordering heuristics. As depicted in Figure 5.12, this improved the performance of the system even further, with the exception of two queries (the length 7 and 14) where our heuristics failed to improve the initial solution. This was caused mainly by the lack of data statistics (primarily the join selectivity of the involved properties) which are necessary for devising a good query plan. After the initial phase during which the mediator collected these statistics, the ordering heuristics were able to also improve the problematic queries, as depicted in Figure 5.13.

5.3 Related Work

There is a long tradition of research on distributed databases in general [92] and distributed query processing in particular [72]. The dominant problem in this field is the generation of an optimal query plan that reduces execution costs as much as possible while guaranteeing the completeness of the result. As described in [72], the choice of techniques for query plan generation depends on the architecture of the distributed system. The paper discusses basic techniques as well as methods for client-server architectures and for heterogeneous databases. Due to our architectural limitations (e.g. limited source capabilities) we focused on join-ordering optimization which can be performed in a centralized manner by the mediator. While some restricted cases of this problem can be solved in a polynomial time [91; 87], the general problem of finding an optimal plan for evaluating join queries has been proved to be NP-hard [96]. The approaches to tackle this problem can be split into several categories [106]: deterministic algorithms, randomized algorithms, and genetic algorithms. Deterministic algorithms often use techniques of dynamic programming (e.g. [91]); however, due to the complexity of the problem they introduce simplifications, which render them heuristic. Randomized algorithms (e.g. [113; 112]) perform a random walk in the solution space according to certain rules. After the stop-condition is fulfilled, the best solution found so far is declared as the result. Genetic algorithms (e.g. [107]) perceive the problem as biological evolution. They usually start with a random population (set of solutions) and generate offsprings by applying crossovers and mutations. Subsequently, the selection phase eliminates weak members of the new population.

Regarding the implementation of a distributed RDF query processing engine, we would like to explicitly mention the research efforts of our colleagues at Vrije Universiteit Amsterdam [1]. Their implementation directly extends the Sesame storage and querying engine [18], relying mainly on its RDF graph merging functionality. The distributed RDF engine uses *Construct* queries to retrieve chunks of RDF graphs form the underlying repositories. In order to answer the initial query, these chunks are temporarily uploaded into the mediator’s memory where they are merged using the Sesame graph API. The current implementation supports only a limited path indexing (single properties) and assumes no
semantic heterogeneity among the sources. As opposed to this approach, the Hera mediator uses a relational view of RDF, distributing *Select* queries to the underlying sources. These sub-queries result in a number of relational tables which are merged in an optimized sequence of join operations. The main reason for which we do not rely on a simple (yet efficient) graph merging is that we assume semantically heterogeneous underlying sources, in which case merging the partial results often requires complex data transformations (specified in the integration model). The Hera mediator supports full path indexing, which allows us to minimize the number of joins at the mediator. On the other hand, the Hera mediator only supports at the moment a subset of the SeRQL query language, while [1] offers the complete query language support.

5.4 Summary of Optimization Techniques

This chapter addressed research question 4. We focused on relational optimization techniques which can be applied in the context of distributed RDF query processing. The intelligent query routing introduced in the previous chapter can be seen as the first optimization step in our framework. The main idea is to maximize the sub-queries that are being pushed to the underlying repositories. This enables the local (instance based) optimization, employs parallelism (the underlying sources work in parallel), and reduces the data traffic by pruning extraneous instances.

To improve the performance of the result assembly, we deployed a combination of two join optimization heuristics: the iterative improvement and the simulated annealing. The former is a greedy heuristics which randomly generates several initial solutions (query plans), taking them as starting points for a walk in the chosen solution space, and accepting those steps that yield improvement. The latter further improves the results by searching the proximity of the sub-optimal solutions, accepting with a certain probability also those moves with higher cost than the current best solution. Finally, in this chapter we also described the implementation of our prototype and provided its performance evaluation.
Chapter 6

RAL: a Graph-based Rdf ALgebra

In this chapter we describe RAL, an RDF algebra for reasoning about queries over RDF data in a (query) language independent fashion. In order to use metadata for application interoperability it is not sufficient to just have a language to describe the metadata. A language for describing queries on that data is also needed. In the XML world there is already a winner in the quest for the most appropriate XML query language, i.e. XQuery [13]. So far the situation is much less clear in case of RDF. Research groups coming from both industry and academia are involved in putting forward several RDF query languages (see the next section). We observe that such query languages often use APIs to describe their semantics. Clearly, for a proper understanding of the languages and a sound theoretical foundation under the languages there is a need for an algebra in the spirit of the one we know from the relational model. Furthermore, since optimization issues are often neglected in existing RDF languages, an algebra for RDF could serve as a platform for finding efficient rewritings of queries. This chapter addresses this need and proposes RAL, an RDF algebra suitable for defining (and comparing) the semantics of different RDF query languages and (at a later stage) for performing algebraic optimizations.

The rest of the chapter is organized as follows. Section 6.1 describes the data model employed with our algebra. Section 6.2 introduces the basic RAL operators, Section 6.3 describes several additional RAL features, and Section 6.4 presents the equivalence laws defined in RAL. Section 6.5 presents related work and the chapter concludes by Section 6.6 which summarizes our arguments for introducing this new algebra and the benefits of RAL.

6.1 RDF Data model

In this section we discuss the data model used with our algebra. We describe how the RDF data structures are represented that are input or output of the expressions formulated in RAL. We start by considering the concept of RDF model.
6.1.1 RDF model

An RDF model is similar to a directed labeled graph (DLG) [74]. However, it differs from a classical DLG since its definition allows for multiple edges between two nodes. It also differs from a multigraph because the different edges between two nodes are not allowed to share the same label. The graph does not necessarily have to be connected and it is allowed to contain cycles.

The nodes in the graph are used to represent resources or literals. Literals (strings) are used to denote content that is not processed further by the RDF processor. The nodes that represent resources can be further classified as nodes representing URI references or blank nodes. URI references are used as universal identifiers in RDF. Each blank node, also called an anonymous resource, is considered to be unique in the graph despite the fact that it has no (explicit) label associated to it. The non-blank nodes are (explicitly) labeled with resource identifiers (URIs) or string values. The edges in the graph represent properties. These edges are labeled by property names. Edges between different pairs of nodes may share the same label, and the same property can be applied repetitively on a certain resource. This RDF feature enables multiple classification of resources, multiple inheritance for classes, and multiple domains/ranges for properties. Both resources and properties are first class citizens in the proposed RDF data model.

We identify the following sets: $\mathcal{R}$ (set of resources), $\mathcal{U}$ (set of URI references), $\mathcal{B}$ (set of blank nodes), $\mathcal{L}$ (set of literals), and $\mathcal{P}$ (set of properties). At RDF level the following holds for these sets: $\mathcal{R} = \mathcal{U} \cup \mathcal{B}$, $\text{rdf:Property} \in \mathcal{U}$, $\mathcal{P} \subset \mathcal{R}$, $\text{rdf:type} \in \mathcal{P}$.

The property $\text{rdf:type}$ defines the type of a particular resource instance. At RDF level any resource can be the target of an $\text{rdf:type}$ property. RDF supports multiple classification of resources, because $\text{rdf:type}$ (as any other property) can be repeated on a particular resource.

**Definition 6.1.** An RDF model $M$ is a finite set of triples (also called statements)

$$M \subset \mathcal{R} \times \mathcal{U} \times (\mathcal{R} \cup \mathcal{L})$$

Each triple or statement in an RDF model contains a resource, a URI reference (which stands for a property), and a resource or literal.

**Definition 6.2.** The set of properties of an RDF model $M$ is

$$P = \{ p \mid (s, p, o) \in M \lor (p, \text{rdf:type}, \text{rdf:Property}) \in M \}$$

The properties in an RDF model are the middle element of a triple in the model, or they are a resource with an $\text{rdf:type}$ property to the $\text{rdf:Property}$ resource.

**Definition 6.3.** Formally the data model (graph model) corresponding to an RDF model $M$ is

$$G = (N, E, l_N, l_E)$$

$$l_N = N \rightarrow \mathcal{R} \cup \mathcal{L}$$

$$l_E = E \rightarrow \mathcal{P}$$
using the following construction mechanism (\(N\) and \(E\) denote the nodes and edges, \(l_N\) and \(l_E\) their labels). For each \((s, p, o) \in M\), add nodes \(n_s, n_o\) to \(N\) (different only if \(s \neq o\)) and label them as \(l_N(n_s) = s\), \(l_N(n_o) = o\), and add \(e_p\) to \(E\) as a directed edge between \(n_s\) and \(n_o\) and label that as \(l_E(e_p) = p\). In the case that \(s\) and/or \(o\) are in \(B\), then \(l_N(n_s)\) and/or \(l_N(n_o)\) are not defined: blank nodes do not have labels.

The function \(l_N(.)\) is an injective partial function, while \(l_E(.)\) is a (possibly non-injective) total function: nodes that have a label have a unique one, edges always have a label but can share it with other edges.

We use quotes for strings that represent literal nodes in order to make a syntactical distinction between them and URI nodes. A URI can be expressed by qualified names (e.g. \(s:\text{Painting}\)) or they can be in absolute form (e.g. \(http://example.com/schema\text{#Painting}\)). Blank nodes do not have a proper identifier which implies that they can be queried only through a property related to them. Compared to XML, which defines an order between subelements, in RDF the properties of a resource are unordered unless they represent items in a sequence container. We remark that not having the burden of preserving element order eases the definition of algebra operators and the associated laws.

6.1.2 Nodes and edges

As we describe in Table 6.1, each node has three basic properties. The \(id\) of a node represents the (identification) label associated to it. The nodes from the subset of resources that represent the blank nodes do not have an \(id\) associated to them. There are two types of nodes: \(\text{rdfs:Resource}\) and \(\text{rdfs:Literal}\). The \(\text{nodeID}\) gives the unique internal identifier of each node in the graph. \(\text{nodeID}\) has the same value as \(id\) for the nodes that have a label, but in addition it gives a unique identifier to the blank nodes. The internal identifier \(\text{nodeID}\) is not available for external use, i.e. it is not disclosed for querying.

<table>
<thead>
<tr>
<th>Basic property</th>
<th>Result for resource (u \in U)</th>
<th>Result for literal (l \in L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id)</td>
<td>(l_N(u))</td>
<td>(l_N(l))</td>
</tr>
<tr>
<td>(type)</td>
<td>(\text{rdfs:Resource})</td>
<td>(\text{rdfs:Literal})</td>
</tr>
<tr>
<td>(\text{nodeID})</td>
<td>internal ID</td>
<td>internal ID</td>
</tr>
</tbody>
</table>

Each edge has three basic properties as described in Table 6.2. Compared with nodes, which have unique identifiers, edges have a \(name\) (label) which may be not unique. There can be several edges sharing the same \(name\) but connecting different pairs of vertices. The \(name\) of an edge is (lexically) identified with the \(id\) of the resource corresponding to the property associated with the edge. The \(subject\) of an edge gives the resource node from which the edge is starting. The \(object\) returns the resource or literal node where the edge ends, i.e. the value of the property.
Table 6.2: Basic properties for edges

<table>
<thead>
<tr>
<th>Basic property</th>
<th>Result for</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge ( p \in P ) from ( r \in R ) to ( o \in R \cup L )</td>
<td>( l_E(p) )</td>
</tr>
<tr>
<td>name</td>
<td>( r )</td>
</tr>
<tr>
<td>object</td>
<td>( o )</td>
</tr>
</tbody>
</table>

**Definition 6.4.** Two non-blank nodes are considered to be equal if they have the same id. Two blank nodes are considered to be equal if they have the same (RDF) properties and the corresponding (RDF) property values are equal.

All non-blank nodes that are considered equal are internally mapped into one node in the graph.

**Definition 6.5.** Two graphs are considered to be equal if they differ only by re-naming the node IDs of their blank nodes.

Note that two graphs for which all their nodes are equal (in terms of node equality) may be not equal themselves (in terms of graph equality) if some corresponding non-blank nodes have different properties and/or different property values.

### 6.1.3 RDFS

RDF Schema (RDFS) [17] provides a richer modeling language on top of RDF. RDFS adds new modeling primitives by introducing RDF resources that have additional semantics (in the previous section we already mentioned \( \text{rdfs:Resource} \) and \( \text{rdfs:Literal} \)). If one chooses to discard this special semantics, RDFS models can be viewed as (plain) RDF models.

The RDFS type system is built using the primitives: \( \text{rdfs:Resource} \), \( \text{rdfs:Property} \), \( \text{rdfs:Class} \), \( \text{rdfs:Literal} \), \( \text{rdfs:subClassOf} \), \( \text{rdfs:subPropertyOf} \), \( \text{rdfs:domain} \), and \( \text{rdfs:range} \). The distinction between \( \text{rdf} \) and \( \text{rdfs} \) namespaces to be used for different resources is more due to historical reasons (RDF was developed before RDFS) than due to semantical ones. Figure 6.1 depicts graphically these RDF/RDFS primitives.

The inheritance mechanism incorporated in RDFS supports taxonomies at class level (using the property \( \text{rdfs:subClassOf} \)) and at property level (using the property \( \text{rdfs:subPropertyOf} \)). It also defines constraints: names to be used for properties, domain and range for properties, etc. These constraints need to be fulfilled by RDF descriptions (later on called instances) in order to validate these instances according to the associated schema.

Every resource that has the \( \text{rdf:type} \) property equal to \( \text{rdfs:Class} \) represents a type (or class) in the RDF(S) type system. Types can be classified as primitive types (\( \text{rdfs:Resource} \), \( \text{rdf:Property} \), \( \text{rdfs:Class} \), or \( \text{rdfs:Literal} \)) or as user-defined types (those are resources
6.1. RDF DATA MODEL

defined explicitly by a particular RDF model to have the rdf:type property equal to rdfs:Class. The type of the resource rdfs:Class is defined reflexively to be rdfs:Class. The resource rdfs:Class contains all the types, which is not the same thing as saying that it includes all the values (instances) represented by these types.

We extend the data model with the set \( C \) (set of classes). At RDFS level the following holds: \( C \subset R, \) rdfs:Resource \( \in C, \) rdf:Property \( \in C, \) rdfs:Class \( \in C, \) and rdfs:Literal \( \in C. \)

**Definition 6.6.** The set of classes of an RDF model \( M \) is

\[
C = \{ c \mid (c, \text{rdf:type}, \text{rdfs:Class}) \in M \}
\]

The most general types are rdfs:Resource and rdfs:Literal which represent all resources and literals, respectively. According to the data model these types are disjoint. Subclasses of the class rdfs:Resource are rdfs:Class and rdfs:Property, rdfs:Class representing all types (already stated above) and rdfs:Property containing all properties. The distinction between properties and resources is not a clear cut one as properties are resources with some additional (edge) semantics associated to them. A property (edge) can be used repetitively between nodes (similar in a way to repeating a particular type in the definition of its instances) which justifies the existence of an *extent* function (defined later on) for properties, as well as for classes. Moreover, property instances can have the rdfs:subPropertyOf property defined in the same way as one can use the rdfs:subClassOf property for classes.

The most important properties (each instance of rdf:Property) are: rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, and rdfs:range. The properties rdfs:subClassOf and rdfs:subPropertyOf are used to define inheritance relationships between classes and properties respectively. Based on the RDF Test Cases [45] rdf:subClassOf and rdf:subPropertyOf can produce cycles, a useful mechanism if we think about class or property equivalence. A resource of type rdf:Property may define the rdfs:domain and the rdfs:range associated to that property: the type of the subject and object nodes of
the property edge. Inspired by ontology languages, like OWL [117], \textit{rdfs:domain} and \textit{rdfs:range} can be multiply defined for one particular property and will have conjunctive semantics.

There is one particular class called \textit{rdfs:Literal} that represents all strings. Note that the RDF Semantics [53] identifies two types of literals: plain literals and type literals. A plain literal is a 2-tuple (lexical form, language identifier) and a typed literal is a 3-tuple (lexical form, language identifier, datatype URI). The datatype URI is an XML Schema datatype [12] or \textit{rdf:XMLLiteral} for XML content. In the data model we simplify the literal definition considering just the character string (the lexical form) for literals. Note that literals are not resources, i.e. one cannot associate properties to them. On the other hand, there are resources that have type \textit{rdfs:Literal} and thus can have properties attached to them. Nevertheless one cannot say which literal this resource denotes. RDF defines also the container classes \textit{rdf:Seq}, \textit{rdf:Bag}, and \textit{rdf:Alt} to model ordered sequences, sets with duplicates, and value alternatives. The properties \textit{rdf:rdf.1}, \textit{rdf:rdf.2}, \textit{rdf:rdf.3} etc. refer to container members.

### 6.1.4 Class and Property nodes

As shown in Table 6.3 each node representing a class has three schema properties. Schema properties associated to nodes are short notations (like a macro) for expressions doing the same computation based only on basic properties. The \textit{type} of a class node is \textit{rdfs:Class}. The set of superclasses (classes from which the current class node is inheriting properties) is given by \textit{subClassOf}. RDFS allows multiple inheritance for classes because \textit{rdfs:subClassOf} (as any other property) can be repeated on a particular class. The \textit{extent} of a class node is the set of all instances of this class.

<table>
<thead>
<tr>
<th>Schema property</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{type}</td>
<td>\textit{rdfs:Class}</td>
</tr>
<tr>
<td>\textit{subClassOf}</td>
<td>$S$ with $S \subset C$</td>
</tr>
<tr>
<td>\textit{extent}</td>
<td>$R'$ with $R' \subset R$</td>
</tr>
</tbody>
</table>

Table 6.3: Schema properties for class nodes

Each node representing a property has five schema properties as shown in Table 6.4. The \textit{type} of a property node is \textit{rdfs:Property}. The set of superproperties (properties which the current property is specializing) is given by \textit{subPropertyOf}. Note that the domain or range of a superproperty should be superclasses for the current property’s domain or range, respectively. The \textit{domain} and \textit{range} return sets of classes that represent the domain and the range, respectively, of the property node. The \textit{extent} of a node is the set of resource pairs linked by the current property: this set of pairs is a subset of the Cartesian product between the associated domain and range extents.

One should note that we assume in the data model that there can be several edges having the same \textit{name} but linking different pairs of resources. All these properties can be
6.2. BASIC RAL OPERATORS

Table 6.4: Schema properties for property nodes

<table>
<thead>
<tr>
<th>Schema property</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>rdf:Property</td>
</tr>
<tr>
<td>subPropertyOf</td>
<td>S with $S \subseteq P$</td>
</tr>
<tr>
<td>domain</td>
<td>D with $D \subseteq C$</td>
</tr>
<tr>
<td>range</td>
<td>R with $R \subseteq C$</td>
</tr>
<tr>
<td>extent</td>
<td>$E$ with $E \subseteq \bigcap_{d \in \text{domain}\text{extent}(d)} \times \bigcap_{r \in \text{range}\text{extent}(r)}$</td>
</tr>
</tbody>
</table>

seen as “instances” (abusing the term “instance” previously referring to resource instances of a particular class) of the property node with the id value equal to their common name.

In absence of a schema, all RDF properties have type rdf:Property, domain $R$, and range $R \cup L$. In this way one can define the extent of an RDF property even if the property is not explicitly defined in a schema. In a schema-less RDF graph all resources are assumed to be of type rdfs:Resource.

6.1.5 Complete models

The RDF Semantics [53] defines the RDF-closure and RDFS-closure of a certain model $M$ by adding new triples to the model $M$ according to a collection of given inference rules. We refer to the original model $M$ as the extensional data and to the newly generated triples as the intensional data. There are two inference rules for RDF-closure and nine inference rules for RDFS-closure. The inference rules for RDF-closure add for all properties in the model the rdf:type property (pointing to rdf:Property). Examples of inference rules for RDFS-closure are the transitivity of rdfs:subClassOf, the transitivity of rdfs:subPropertyOf, and the rdf:type inference for an rdf:type edge that follows after an rdfs:subClassOf edge. One should note that the resulting output of applying these inference rules may trigger other rules. Nevertheless the rules will terminate for any RDF input model $M$, as there is only a finite number of triples that can be formed with the finite vocabulary of $M$.

Definition 6.7. An RDF model $M$ is complete if it contains both its RDF-closure and RDFS-closure.

In the proposed data model we consider complete models and we neglect reification and the properties rdfs:seeAlso, rdfs:isDefinedBy, rdfs:comment, and rdfs:label without losing generality.

6.2 Basic RAL operators

The purpose of defining RAL is twofold: to provide a reference mathematical study for RDF query languages and to enable algebraic manipulations for RDF query optimization. RAL is an algebra for RDF defined from a database perspective, some of its operators
being inspired by their relational algebra counterparts. We used a similar approach in developing XAL [38], an algebra for XML query optimization.

During the presentation of RAL operators we will use the RDF data from the example in Figure 6.2 as input for the operators. It is assumed that all operators know about the complete RDF model as it was defined in Definition 6.7. That means that they all have the complete knowledge (both extensional and intensional data) present in the given model. Variants of the proposed operators can be defined using the suffix “ˆ” which will make the operators neglect the intensional data, i.e. data derived by applying RDF(S) inference rules to the input model is neglected (similar to RQL’s “strict interpretation”).

Figure 6.2 is an excerpt from the RDF schema and RDF instance of some Web data describing different painting techniques. For reasons of simplicity we consider only one painting technique (“Chiaroscuro”), one painter (“Rembrandt”), and two paintings of the same painter (“StoneBridge” and “SelfPortrait”). The figure does not present the RDFS primitives rdfs:Resource, rdf:Property, rdfs:Class, and rdfs:Literal from which all the resources and literals are derived. In order to simplify Figure 6.2 we chose to present only the extensional data and just one intensional data element given by the inferred edge rdf:type between r4 and Creator. For the same reasons we omit from the figure edges representing the inverse properties exemplifies and painted_by between instances (e.g. the edge labeled painted_by between r2 and r4) that are nevertheless part of the data model.

We define RDF collections to be sets of nodes (resources/literals). A collection is
denoted by \( \{e_1, e_2, \ldots, e_n\} \), where \( e_1, e_2, \ldots, e_n \) are the nodes in the collection. A node, a unique element in the RDF graph, is also a unique element in a collection that contains it. All RAL operators are closed for collections, which implies that RAL expressions can be easily composed. The collection concept is similar to the monad concept from mathematics [123]. A monad is defined over a certain type \( M \). In contrast to the monad, RAL collections are more liberal in the sense that they are not restricted to a particular type \( M \). A RAL collection can contain both literals and resources of different types. A monad is defined as a triple of functions \((\text{map}^M, \text{unit}^M, \text{join}^M)\). RAL also has the \text{map} operation defined and the monad \text{join} operation is equivalent to RAL’s \text{union} operation. In RAL there is no \text{unit} operation as the singleton collection \( \{n\} \) is written in the same way as the single node \( n \). Based on the similarities between monads and RAL collections, one can reuse the three monad laws (left unit law, right unit law, and associativity law) as equivalence rules in RAL (see the first three RAL laws from Section 6.4). The fact that RAL collections are not ordered enables the commutativity law of some binary operations (see Law 6.11 from Section 6.4). Compared to the relational algebra, RAL is more powerful as binary operations like union do not have to meet the “compatibility” condition from the relational algebra.

RAL operators come in three flavors: extraction operators retrieve the needed resources from the input RDF model, loop operators support repetition, and construction operators build the resulting RDF model. The RAL philosophy is based on the fact that the collection of nodes represents a collection of graph components that contain these nodes. Using the extraction operators a subgraph of the original graph is selected. The construction operators build a new model by creating nodes/edges as well as reusing old nodes (possibly without some edges) and old edges.

The general form of the operators is

\[
o[f](x_1, x_2, \ldots x_n : \text{expression})
\]

Informally, this form represents the following. For each binding of \( x \) to a tuple from the input collections, \( f(x) \) is computed. A tuple is formed by taking one element from each input collection: \( x_1, x_2, \ldots x_n \). Note that \( x_1, x_2, \ldots x_n \) are algebra expressions that return collections. \( f \) is a function that may use basic/derived properties or one of the proposed operators. Based on the semantics of operator \( o \) a partial result for the application of \( o \) to \( f(x) \) is computed for each binding \( x \). The operator result is obtained by combining (through set union) all partial results. All unary operators use this implicit union mechanism, the \text{map} operator, to compute the result. In the operator’s general form, the function \( f \) is optional. For readability reasons we use for binary operators the infix notation.

RAL operators are defined to work on any RDF description, with or without an explicit schema. Note that implicitly there is always a default schema based on the following RDFS primitives: \texttt{rdf:Resource}, \texttt{rdf:Property}, \texttt{rdf:Class}, and \texttt{rdf:Literal}. These RDFS primitives can be used to retrieve a particular schema in case that such information is not known in advance. Once the application schema is known, one can formulate queries to return instances from the input model.
6.2.1 Extraction operators

The extraction operators retrieve the resources/literals of interest from the input collection of nodes. If the operator is not defined on nodes that represent literals, these nodes are simply neglected.

In the examples that illustrate the operators we will use expressions that return collections of resources from the example RDF model $m$ of Figure 6.2. The expression $c$ represents the collection (set) of all resources present in model $m$.

**Projection**

$$\pi[re\_name](e : expression)$$

The input of the projection is a collection of nodes (specified by the expression $e$) and the projection operator computes the values (objects) of the properties with a name given by the regular expression $re\_name$ over strings. The symbol $\#$ represents the wildcard that matches any string.

**Example 6.1.** $\pi[exemplified\_by](r1)$ returns the collection of artifacts that exemplify the painting technique $r1$ from the input model (depicted in Figure 6.2): $r2$ and $r3$.

**Example 6.2.** $\pi[(P|p)a\_int[s]\#](r4)$ returns the collection of paintings painted by $r4$: $r2$ and $r3$.

**Example 6.3.** $\pi[rdf\_type](r4)$ returns the collection of resources representing a type of $r4$: Painter, Creator, and rdf:Resource.

**Selection**

$$\sigma[condition](e : expression)$$

In a selection the condition is a Boolean function that uses as constants URIs and/or strings. The operators allowed in the condition are RAL operators, the usual comparison operators ($=, \geq, \leq, <, >, <\geq$), and logical operators (and, or, not). The input of the selection is a collection of nodes and the operator selects only the nodes that fulfill the condition.

**Example 6.4.** $\sigma[\pi[t\_name] = "Chiaroscuro"](c)$ is a selection operation applied to the collection $c$ of all resources in the input model. The expression returns the resource(s) representing the painting technique with the name “Chiaroscuro” (i.e. $r1$).

**Example 6.5.** $\sigma[\pi[rdf\_type] = Creator](\{r3, r4\})$ returns resources from the input model with the value of rdf:type being Creator: $r4$, since $r4$ is a resource of type Painter and Painter is a subclass of Creator.
Example 6.6. $\sigma [\pi [\text{rdf:type}] = \text{Creator}] (\{r3,r4\})$ (different from the selection in the previous example, as "" implies the use of only the extensional data) returns the empty collection, as the inferred rdf:type of r4 (i.e. Creator) from the input model will not be available to the operator.

Cartesian product

$$(x : \text{expression}) \times (y : \text{expression})$$

The Cartesian product takes as input two collections of nodes on which it performs the set-theoretical Cartesian product. Each pair of nodes is used to build an anonymous resource that has all the properties of the original resources. Thus, this newly built resource will have all the types of the original two resources (RDF multiple classification of resources). The final output is the collection of all those anonymous resources.

Example 6.7. $\sigma [\pi [\text{rdf:type}] = \text{Technique}] (c) \times \sigma [\pi [\text{rdf:type}] = \text{Painter}] (c)$, where $c$ represents the collection of all resources in the input model, returns one anonymous resource having all the properties of the only technique r1 and the only painter r4. As a consequence this anonymous resource has both types Technique and Painter.

Join

$$(x : \text{expression}) \bowtie [\text{condition}] (y : \text{expression})$$

The join expression is defined to be a Cartesian product followed by a selection, so equivalent to

$$\sigma [\text{condition}](x \times y)$$

The expression has as input two collections of resources that have their elements paired only if they fulfill the condition (referring to the left and right operands). Anonymous resources are built for each such pair. The output is the collection of all those anonymous resources.

Example 6.8. $(t := \sigma [\pi [\text{rdf:type}] = \text{Technique}] (c)) \bowtie [\pi [\text{exemplified by}](t) = \pi [\text{paints}](p) | (p := \sigma [\pi [\text{rdf:type}] = \text{Painter}] (c))$, where $c$ represents the collection of all resources in the input model, returns an anonymous resource having all the properties of r1 and r4. Note that in this expression r1 and r4 are paired because there is a painting (e.g. r2) that exemplifies r1 and is painted by r4.

Union

$$(x : \text{expression}) \cup (y : \text{expression})$$

The union operator combines two input collections of nodes reflecting the set-theoretical union.
CHAPTER 6. RAL: A GRAPH-BASED RDF ALGEBRA

Difference

$$(x : expression) - (y : expression)$$

The difference operator returns the nodes present in the first input collection but not in the second input collection.

Intersection

$$(x : expression) \cap (y : expression)$$

The intersection operator returns the nodes present in both input collections.

6.2.2 Loop operators

Loop operators are used in RAL to control the repetitive application of a function or operator. They express repetition at input and/or function/operator level.

Map

$$map[f](e : expression)$$

The map operator is defined as

$$\cup(f(e_1), f(e_2), ... f(e_n))$$

if the collection $e$ contains the elements $e_1, e_2, ... e_n$. So, the map operator expresses repetition at input level. The results of applying the function/operator $f$ to each element in the input collection are combined (through set union) to obtain the final result. All unary extraction operators have an implicit map operator associated with them.

Example 6.9. $map[id](c)$, where $c$ represents the collection of all resources in the input model, computes the labels of all the non-blank nodes in the input model, i.e. the labels of all resources having an id property.

Kleene star

$$*[f](e : expression)$$

The Kleene star operator is defined as

$$e \cup f(e) \cup ... f(f(...(f(f(e)))...) \cup ...$$
So, the Kleene star operator expresses repetition at function/operator level. It repeats the
application of the function/operator \( f \) on the given input for possibly an infinite number
of times. For each iteration the result is obtained by combining (through set union) the
output of applying the function/operator on the input with the input. If after an iteration
the result is the same as the input, a fixed point is reached and the repetition stops. In order
to ensure termination, a variant of this operator that specifies the number of iterations \( n \)
is defined below:

\[ *[f, n](e : expression) \]

Note that the map operator does not include the input in the result, while the Kleene
star operator does.

**Example 6.10.** \( \text{map[id]}(\*[\pi[rdfs:subClassOf]](\text{Painting})) \) gives the id of all ancestor
classes in the type hierarchy starting with Painting. For our example the result will contain
three labels denoting the types Painting, Artifact, and rdfs:Resource. If there would have
been loops made by the rdfs:subClassOf property in the input model, the above example
would still have terminated. The fact that the input model has a finite number of classes
implies that at a certain moment a fixed point is reached (we obtain the same output
collection as for the previous iteration) and thus the Kleene star operator terminates.

### 6.2.3 Construction operators

Querying an RDF model implies not only extracting interesting nodes from the input model
but also constructing an output model by deleting nodes/edges from the extracted graph
and by creating new nodes/edges.

Before actually committing a construction operation, the RDF constraints are checked
on the output model. If these constraints are not met, the operation aborts. Examples of
RDF constraints are: resource identifiers have to be unique, the value of rdf:type cannot
be a literal, literals cannot have properties etc.

**Create node**

\[ \text{cnode[type, id]}() \]

The create node operator possibly adds a new node to the graph. The input collection
is not used in the operator semantics. The type of the new node, specified by type, is
a resource of type rdfs:Class. The id is a resource identifier if the node represents a
resource, or a string if the node represents a literal. The id is used as input in the system’s
new id generator (nig) skolem function. This function returns the unique nodeID. The
nodeID is equal to id if id is given, or it is a new unique identifier if id is empty. In the
first case an old node identifier is returned if id is already used as a nodeID in the data
model. In the second case a blank node is assigned a new nodeID. Note that the function
nig is injective. As a side effect of this operator, an edge representing the type property
is added between the newly created resource and its associated type resource. The create
node operator returns the created node (a collection containing one node).

Example 6.11. \texttt{cnode[Painter]()} creates a blank node of type Painter, while
\texttt{cnode[Literal,"Caravagio"]()} creates a Literal node representing the string “Caravagio”.

Create edge

\[ \texttt{cedge[name, subject]}(\text{object : expression}) \]

The create edge operator possibly adds new edges (properties) to the graph. The name
(label) of the edges, as specified by name, is the id of a resource of type \texttt{rdf:Property} (the
id of a property resource). The subject and the object must have types complying with
the domain and the range of the property resource indicated by name. If there is already
an edge between subject and object with the label given by name then there is no need
to create a new edge. Recall that the RDF semantics does not allow the presence of two
edges that share the same label between the same two nodes.

The subject is one node (or singleton collection) in the graph. The object can be
a collection of nodes. Note that in the above description object denotes a node from
the input collection. The edges are created between the subject node and the object
node(s). The create edge operator returns the subject node (a collection containing one
node). This operation can be generalized after introducing variables in RAL as shown in
Subsection 6.3.1.

Example 6.12. If \( n_1 \) and \( n_2 \) are the two nodes constructed in Example 6.11, \( n_1 \) denoting
the blank node and \( n_2 \) denoting the literal node, \texttt{cedge[name, n1]}(\texttt{n2}) creates an edge labeled
name between the nodes \( n_1 \) and \( n_2 \).

Delete node

\[ \texttt{dnode(e : expression)} \]

The delete node operation deletes nodes from the graph. The input collection gives the
nodes that are removed, and the operation returns the empty collection. As a side effect,
the edges connected to these nodes as subject or object are also deleted.

Example 6.13. \texttt{dnode(\{r2, r3\})} deletes the nodes \( r_2 \) and \( r_3 \), and all the edges connected to
\( r_2 \) or \( r_3 \). For the given model this implies the elimination of the two resources representing
paintings and their associated edges.
6.3. ADDITIONAL RAL FEATURES

Delete edge

dedge[re_name, subject](object : expression)

The delete edge operation deletes edges from the graph. The edges that are deleted have to start in the subject node and to end in one of the nodes from the object collection. The name (label) of the edges to be deleted is given by the regular expression re_name, a regular expression over strings. If the subject and/or the object expressions are empty, the edges to be deleted are identified by the remaining input arguments. The operation returns the subject input.

Example 6.14. dedge(#, r1)(\{r2, r3\}) deletes the edges between r1 and r2, and between r1 and r3, irrespective of their name. In the concrete example the information that two paintings exemplify the painting technique (r1) is removed.

6.3 Additional RAL features

6.3.1 Variables

A variable is a substitute for a collection of nodes (possibly) resulting from an evaluation of an algebraic expression. A variable thus serves as a shortcut of such an expression that can be used in more complex algebraic expressions. There are several reasons for introducing variables. First, as we already saw in the definition of the join operator, the join’s selection condition may need a reference mechanism for the two operands (input collections). Second, variables can be very useful in expressing complex expressions in which a collection is used repeatedly. The third reason is related to the fact that query languages like RQL give their results in terms of a table that has as columns variables and as rows bindings of these variables. If one would like to use RAL to implement RQL expressions, this compatibility feature should be met.

Example 6.15. y := π[paints](x := r4) instantiates x with r4 and y with r2 and r3. If one wants to export these variables, the result will be a table, similar to a table returned by RQL, with two columns x and y, and two rows: the first row contains r4 and r2 and the second row contains r4 and r3.

The last reason for having variables is the fact that it has a nice application for the construction operators. If the extracted nodes are bound to variables, these variables can be elegantly used in the construction part of RAL. The create edge operation can be extended by allowing a collection of nodes not only in the object part but also in the subject part by representing both parts with variables. The semantics of this construction operator is that for each variable binding an edge will be created between the corresponding nodes.

Example 6.16. Consider the variable bindings from the previous example, y := π[paints] (x := r4). The expression cedge[peind, x](y) will add two edges with the label peind (the
French translation of paints) to the model, one between r4 and r2, and one between r4 and r3.

![Diagram](image)

Figure 6.3: Variable bindings.

As shown in the previous example the value of the inner variable \( x \) is associated with two values of the outer variable \( y \). The two pairs \((r4,r2)\) and \((r4,r3)\) created by the projection operator can be seen as two 2-tuples similar to those from the relational model.

Generalizing this we can say that n-1 nested projections create a set of sets of sets ... of sets (n times) of variable bindings or in other words they generate n-tuples.

**Example 6.17.** To illustrate the above consider the tuple bindings for the following expression operating over the RDF graph depicted in Figure 6.3: \( z := \pi[e3](y := \pi[e2](x := x1)) \). The resulting bindings are the following 3-tuples: \((x1,y1,z1)\), \((x1,y1,z2)\), \((x1,y1,z3)\), \((x1,y2,z3)\), and \((x1,y2,z4)\).

Note that by generating these tuple bindings we possibly generate duplicates at the variable level (the variable \( x \) is bound five times to the same value \( x1 \) in the above example). These duplicates are removed prior to applying variable bindings as input for an operator in order to assure “duplicate-free” collections on which our operators are closed.

In order to be able to compare results with RDF query languages that use as their output tables of tuples, RAL provides a mechanism to export tuple bindings. This is achieved simply by specifying the variable names participating in the tuple, separated by “,”. For instance \( x,y,z \) exports the five tuples from the previous example. Note that if we export only one variable, say \( x \), there will still be five 1-tuples (five times \( x1 \)), i.e. export does not remove duplicates.

So far we discussed only variables which were bound during the multiple application of the projection operator, i.e. they occurred on the same path in the graph. These variables are dependent in the sense that the value of the next variable(s) depends on the binding of the previous ones. There might be, however, variables that do not depend on each other, i.e. they do not appear on the same path in the graph. In case of exporting independent variables, export performs a cross product of their bindings.
**Example 6.18.** The variable \( p \) from \( p := \pi[e_1](x_1) \) is independent from the variables introduced in the previous example. Exporting \( p, y, z \) results in the following tuple bindings: \((p_1,y_1,z_1), (p_1,y_1,z_2), (p_1,y_1,z_3), (p_1,y_2,z_4), (p_2,y_1,z_1), (p_2,y_1,z_2), (p_2,y_1,z_3), (p_2,y_2,z_3)\), and \((p_2,y_2,z_4)\).

### 6.3.2 Additional Operators

**Sort**

\[ \Sigma[value\_expression(e)](e : expression) \]

The sort operator orders alphabetically a collection based on a value given by \( value\_expression \). This \( value\_expression \) is an expression that returns a collection of strings (literals or URI references). The \( value\_expression \) is applied for each node in the input collection and the original nodes are ordered alphabetically based on the computed values.

Note that RAL collections are sets, i.e. they are not ordered. Nevertheless it is useful to be able to output ordered collections, as a last operator to be possibly used in a RAL expression.

**Example 6.19.** \( \Sigma[\pi[name]](\pi[paints](r_4)) \) orders alphabetically the resources representing \( r_4 \)'s paintings based on their names.

### 6.3.3 RQL and RAL

RQL [63] is one of the most advanced RDF(S) query languages to date and RAL was designed taking into consideration RQL’s power of expression. RQL path expressions from the FROM clause and RQL conditions from the WHERE clause can easily be converted in RAL expressions using RAL operators. The vice versa conversion is not always possible as there are RAL expressions (e.g. expressions with construction operators) that are not expressible in RQL. Unlike RAL, RQL is not a closed query language; it takes as input an RDF graph and it returns a table of variable bindings. Since this table does not represent an RDF graph (just values of some variables) it cannot be used again as input for the next query. As a consequence, views are not supported. Nevertheless, RQL offers some degree of nesting queries in the FROM and WHERE clauses.

**Example 6.20.** Find the name of all painting techniques and the name of the painters who used these techniques. In RQL this query looks as follows:

```
SELECT Xtn, Zcn
FROM {X:Technique}exemplified_by.painted_by{Z}.cname{Zcn},
    {M}tname{Xtn}
WHERE X=M
```
In our concrete example this query returns two identical rows. The pair Chiaroscuro, Rembrandt appears twice as a result since there are two paintings (r2 and r3) that exemplify the Chiaroscuro technique and are painted by Rembrandt (r4).

The following RAL program exports the same variable bindings of Xtn and Zcn as the above RQL query:

\[
\begin{align*}
  z &:= \pi[painted\ by](\pi[exemplified\ by](x := \sigma[\pi[rdf\:type] = \text{Technique}] (e))); \\
  Xtn &:= \pi[tname](x); \\
  Zcn &:= \pi[cname](z); \\
  Xtn, Zcn
\end{align*}
\]

Instead of just outputting variable values in a table-like fashion the construction operators of RAL allow for constructing a full-fledged RDF graph. For instance the following expression connects all painters from the previous query to the techniques they were using by adding a ptechnique edge: \(\text{edge|ptechnique, z}[(x)]\).

### 6.4 RAL equivalence laws

One of the advantages of using an algebra expression to express a query is the ability to rewrite this expression in a form that satisfies certain needs. For example, an automatic translator from RQL to RAL can use RAL equivalence laws to rewrite algebra expressions for query optimization purposes.

The proposed set of equivalence laws is inspired by the monad laws [123], and the relational algebra’s equivalence laws [115]. In [5] it was shown how relational equivalence laws can be reused (redefined) in an object oriented context.

**Law 6.1 (Left unit).** If \(e_1\) is of unit type (singleton collection), i.e. \(e_1 = \{n\}\), then

\[e_2(e_1) = e_2(n)\]

**Law 6.2 (Right unit).** If \(e_2\) is the identity function, i.e. \(e_2(e) = e\), then

\[e_2(e_1) = e_1\]

**Law 6.3 (Empty collection).** If \(e_2\) is the empty function, i.e. \(e_2(e) = ()\), then

\[e_2(e_1) = ()\]

**Law 6.4 (Decomposition of \(\bowtie\)).**

\[e_1 \bowtie [\text{condition}] e_2 = \sigma[\text{condition}](e_1 \times e_2)\]

**Law 6.5 (Decomposition of \(\pi\)).** If name is a regular expression that can be decomposed in several regular expressions name\(_1\), ... name\(_n\) then

\[\pi[\text{name}](e) = \pi[\text{name}\_1](e) \cup ... \pi[\text{name}\_n](e)\]
6.4. RAL EQUIVALENCE LAWS

Law 6.6 (Cascading of \(\sigma\)).
\[
\sigma[c_1 \land \ldots c_n](e) = \sigma[c_1](\sigma[c_n](\ldots(\sigma[c_n](e))\ldots))
\]

Law 6.7 (Commutativity of \(\sigma\)).
\[
\sigma[c_1](\sigma[c_2](e)) = \sigma[c_2](\sigma[c_1](e))
\]

Law 6.8 (Commutativity of \(\sigma\) with \(\pi\)). If the condition \(c\) involves solely nodes that have incoming edges named by the regular expression \(\text{name}\), then
\[
\pi[\text{name}](\sigma[c(\pi[\text{name}])](e)) = \sigma[c](\pi[\text{name}](e))
\]

Law 6.9 (Commutativity of \(\sigma\) with \(\times\)). If the condition \(c\) involves solely nodes from \(e_1\), then
\[
\sigma[c](e_1 \times e_2) = \sigma[c](e_1) \times e_2
\]

Law 6.10 (Commutativity of \(\sigma\) with \(\cup\), \(\cap\), \(\neg\)). If \(\theta\) is one of the operators \(\cup\), \(\cap\), and \(\neg\), then
\[
\sigma[c](e_1 \theta e_2) = \sigma[c](e_1) \theta \sigma[c](e_2)
\]

Law 6.11 (Commutativity of \(\cup\), \(\cap\), \(\times\)). If \(\theta\) is one of the operators \(\cup\), \(\cap\), and \(\times\) then
\[
e_1 \theta e_2 = e_2 \theta e_1
\]

Law 6.12 (Commutativity of \(\pi\) with \(\times\)). If \(\text{name}\) is a regular expression that can be decomposed in two regular expressions \(\text{name}_1\) and \(\text{name}_2\), and if \(\text{name}_1\) involves solely nodes in \(e_1\), and \(\text{name}_2\) involves solely nodes in \(e_2\), then
\[
\pi[\text{name}](e_1 \times e_2) = \pi[\text{name}_1](e_1) \times \pi[\text{name}_2](e_2)
\]

Law 6.13 (Commutativity of \(\pi\) with \(\cup\)).
\[
\pi[\text{name}](e_1 \cup e_2) = \pi[\text{name}](e_1) \cup \pi[\text{name}](e_2)
\]

Law 6.14 (Associativity of \(\cup\), \(\cap\), \(\times\)). If \(\theta\) is one of the operators \(\cup\), \(\cap\), and \(\times\) then
\[
(e_1 \theta e_2) \theta e_3 = e_1 \theta (e_2 \theta e_3)
\]

In order to illustrate the usefulness of the above laws for query optimization we use an example. The query optimization heuristics is based on pushing the selections/projections down as far as possible and applying the most restrictive selections first as it was done similarly in the relational algebra context. The example schema is given in Figure 6.4. It is a slightly modified example compared to the one from Figure 6.2 in the sense that the properties between concepts are replaced by literal (value) properties that function as concept identifier locators. This new example comes from a Web data integration exercise in which different schemata need to be merged “by value”. We chose this schema example
as it better (compared with the example from Figure 6.2) illustrates the proposed query optimization.

The query under investigation is: Return in alphabetical order the nationalities of the painters that used the Chiaroscuro painting technique. A query parser will produce the initial query tree given in Figure 6.5. In all query trees $a$ represents the collection of all resources in the input model classified under the schema from Figure 6.4.

A query execution module will process a node in a query tree as soon as the operands are available. Such a node will be replaced by the collection that results from executing the node’s associated expression. The execution terminates when the root node is processed. The final query result is the collection obtained from processing the root node.

In the example, during the execution of the initial query tree a very large Cartesian product between all painters, paintings, and techniques is generated. By pushing the selections down (using Law 6.6, Law 6.7, and Law 6.9) one can get the query tree in Figure 6.6.

A further improvement is obtained by applying the most restrictive selections first (using Law 6.6, Law 6.7, Law 6.9, Law 6.11, and Law 6.14). The resulting query tree is given in Figure 6.7. So, with the aid of RAL laws three equivalent query trees were obtained.

In order to better understand why it is more efficient to execute the last query tree,
we will give a quantitative dimension to our example. Suppose that the instance of the proposed schema example has 5 painting techniques, 100 painters, and 1000 paintings. Only 100 of all paintings use the Chiaroscuro painting technique. Let’s compute now the number of elements generated by the Cartesian products for each query tree. For the first query tree we have $100 \times 1000 + 5 \times 100 \times 1000 = 600,000$ elements, for the second query tree $100 \times 1000$ (painters are matched to their paintings) + $1000 \times 1$ (paintings are matched to the Chiaroscuro painting technique) = $101,000$ elements, and for the last query tree $1 \times 1000$ (paintings are matched to the Chiaroscuro painting technique) + $100 \times 100$ (paintings that use the Chiaroscuro technique are matched to their painters) = $11,000$ elements. The most efficient to execute is the last query tree as its Cartesian products produce the smallest number of elements.

6.5 Related work

In the previous section we addressed the role of an algebra for the definition and comparison of query languages and for query optimization. At present, there already exist a few RDF query languages but to our knowledge there is no full-fledged RDF algebra. The only algebraic description of RDF that we encountered at the time of our research, is the RDF data model specification described in [84]. This specification is based on triples and it provides a formal definition of resources, literals, and statements. Despite being nicely defined, the specification does not include URIs, neglects the RDF graph structure, and does not provide operations for manipulating RDF models. Another formal approach, which aims not only at formalizing the RDF data model but also at associating a formal semantics to it, is the RDF Semantics (RS) [53]. However, it does not qualify as an algebraic
As implementation of RDF toolkits started before having an RDF query language, there are a lot of RDF APIs present today. Three main approaches for querying RDF (meta)data have been proposed.

The first approach (supported in the W3C working group by Stanford) is to view RDF data as a knowledge base of triples. Triple [104], the successor of SiLRI (Simple Logic-based RDF Interpreter) [29], maps RDF metadata to a knowledge base in Horn Logic (replacing Frame Logic). A similar approach is taken in Metalog [80], which matches triples to predicates in Datalog, a subset of Horn Logic. In this way one can query RDF descriptions at a high level of abstraction: the querying takes place at a logical layer that supports inference [48].

The second approach (proposed by IBM) builds upon the XML serialization of RDF. In the “RDF for XML” project, an RDF API is proposed on top of the IBM AlphaWork’s XML 4 Java parser. In the context of the same project a declarative query language for RDF (RDF Query) [79] was created for which both input and output are resource containers. One of the nice features of this query language is that it proposes operators similar to the relational algebra, leaving the possibility to reuse some of the 25 years experience with relational databases. Unfortunately, the language fails to include the inference rules specific to RDF Schema, losing description semantics.

This second approach is taken even further in [71], where RDF query and transformation languages are proposed that extend existing XML technologies. Similarly to XPath, in [71]
6.6. **RAL Summary**

RDFPath is defined for locating information in an RDF graph. The location step and the filter constructs were present also in XPath, but the primary selection construct is new in this language. With the RDF graph being a forest, one needs to specify from which trees the selection will be made. RDFT is an RDF declarative transformation language à la XSLT [65], while RQuery, an RDF query language, is obtained by replacing XPath [7] with RDFPath in XQuery [13]. However, this approach is not using the features specific for RDF, as the RDF Schema is being completely neglected.

The third approach uses the RDF Graph Model for defining the RDF query language RQL [63]. It extends previous work on semistructured query languages (e.g. path expressions, filtering capabilities etc.) [24] with RDF peculiarities. Its strength lies in the ability to uniformly query both RDF descriptions and schemata. Compared to the previous approach it exploits the inference given in the RDF Schema (e.g. multiple classification of resources, taxonomies of classes and properties, etc.) making it one of the most advanced RDF query language proposed so far.

Other query languages for RDF have been proposed during the last years: we name Algae [93] (W3C) and rdfDB Query Language [47] (Netscape) as graph matching query languages. RDF query languages similar to rdfDB Query Language are: RDFQL [56], David Allsop’s RDF query language [2], SquishQL [86], and RDQL [102] (HP Labs) an implementation of SquishQL on top of the Jena RDF API [82] of Brian McBride (HP Labs). Some other proposed RDF APIs are: Wilbur [73] (Nokia), the RDF API introduced in [85], and Redland [4].

A characteristic aspect of most existing RDF query languages is that they disregard the (re)construction of the output, leaving the output as a “flat” RDF container of resources. The focus is on the extraction of the proper resources for the given query, not on building a new RDF data structure. A notable exception to this is the successor of RQL: Sesame Rdf Query Language (SeRQL) [19]. This query language, similarly to RAL, takes into account also the construction part: deriving from the input data structure a new RDF data structure. The resulting RDF graph can contain new vertices and edges not present in the original RDF graph. To express RDF queries, both the extraction and construction parts should be covered. This construction part is not only necessary for querying, but it is also useful for query optimization and potentially extends the scope of the query language to a view definition language\(^1\).

---

\(^1\)A view definition language also has to include a view naming scheme, which SeRQL currently lacks.
and also serve as a platform for finding efficient rewritings of queries.

In this chapter we proposed RAL, one of the first RDF algebraic formalisms developed from a database perspective. RAL is built on top of a data model which takes the form of a directed-labeled graph, where the graph nodes represent resources and the edges denote properties. The RAL operators defined over this model take as input a collection of nodes. From this point of view RAL is node centric. However, these (explicit) nodes serve as entry points to the (implicit) data of the underlying data model, which can be obtained by (recursive) traversal of the input nodes’ properties. So in fact, every node in the collection potentially represents an (implicit) graph. Similarly to the input, also the output of the RAL operators takes the form of a collection of nodes. This makes RAL closed under composition which is an important property for defining complex nested queries.

The set of proposed RAL operators includes extraction, repetition, and construction features. The extraction operators serve mainly for querying purposes; they retrieve the resources/literals of interest from the input collection of nodes.

The loop operators are used in RAL to control the repetitive application of a function or operator. They express repetition at input and/or function/operator level.

The construction operators include constructing new nodes and edges and deleting the old ones. This facilitates the definition of RDF graph transformations. Other features of RAL include a variable binding mechanism, a (relational) export facility, a sorting function etc.

RAL also defines a number of equivalence laws that facilitate query, rewriting which is the essential prerequisite for algebraic optimizations. The queries are expressed in equivalent query plans which are compared in terms of evaluation costs, choosing the least expensive one.

The Rdf Algebra defined in this Chapter contributes to the answer for Research question 4. The proposed algebraic formalism allows to reason about high-level RDF queries in terms of their query plans, facilitating their rewriting and further optimization.
Chapter 7

User Interface: Browsing and Query Formulation

Since RDF(S) is an acknowledged backbone of the Semantic Web architecture, it is natural that it deserves to have a human readable interface which could be used for browsing and composing queries. In this chapter we focus on the starting point of the Hera pipeline: the initial formulation of the user information need and the user interface which helps to transform this information need into the syntax of a particular RDF query language.

We observe that the work that has been done so far (or its results) on actual creation of semantic annotations of Web resources is still in its infancy. However, this work has to start incrementally in small communities related to specific domains. These communities should agree (or have already agreed) on the terms they are using and find the right motivation to write them down formally as ontologies, and allow others to benefit from them as well. The more people will be using these ontologies the more popular they will become. They will evolve and interconnect with other "neighboring" ontologies, eventually creating something that one could call the emerging Semantic Web. So, apart from creating ontologies, one of the essential prerequisites to make the Semantic Web happen is to also convince the end-users to take advantage of the already existing ontologies for accomplishing their tasks. To do so, they must be offered a whole framework of tools wrapped nicely by a user interface that is both expressive and comprehensible enough. Here it is where we see the major gap between what is currently available and what the end-user desires.

The rest of the chapter is structured as follows. Section 7.1 introduces our motivating example. Section 7.2 reviews the existing user interface metaphors for displaying RDF(S), concluding that due to RDF(S) peculiarities a new interface metaphor is needed to convey RDFS-based ontologies to the end-user in a browseable and comprehensible form. Section 7.3 addresses this issue by proposing the EROS interface. The section details both the class-centric and the property-centric view of EROS, and explains how this interface can help the user with the query formulation. Section 7.4 contains a brief summary of the chapter.
CHAPTER 7. USER INTERFACE: BROWSING AND QUERY FORMULATION

7.1 The Challenge in Visualizing RDFS Ontologies

In this section we briefly review the main features of RDFS that may be of concern for the user interface design and introduce our ontology example on which we demonstrate the different user interface approaches. As we motivated above, there is an apparent need for a suitable user interface for browsing and querying RDFS-based ontologies. The challenge in designing such a user interface lies in the fact that it is difficult to show the whole expressive power of RDFS and at the same time to keep the user interface still comprehensible, easy to use, browse, and pose queries.

As mentioned earlier, RDF Schema is an extension of RDF which facilitates the creation of light-weight ontologies that describe at type level the underlying RDF instances. RDFS defines a modeling language by assigning a special semantics to several (system) resources and properties. The transitive property \texttt{rdfs:subClassOf} (\texttt{rdfs:subPropertyOf}) provides an inheritance-like mechanism for building hierarchies of classes (properties). A class (property) is allowed to have several of these properties, which means that it can have several super classes (super properties), which facilitates the multiple inheritance mechanism. The properties are associated with classes only by specifying their \texttt{rdfs:domain} and \texttt{rdfs:range} (unlike in the case of object modeling when properties are embedded in an object). This facilitates the defining of properties separately from classes, making them first-class citizens in the modeling language. In our search for a suitable interface, we focused mainly on RDFS (not RDF instances) as this is the core formalism used to denote the CM and other models in the Hera framework.

To illustrate the peculiarities of RDFS from the user interface point of view, throughout this chapter we use an example of an ontology which was developed in a community of photo-enthusiasts. The ontology describes a photo stock library that facilitates semantic annotation of stored images in order to allow their later access by expressing queries over the annotation ontology. Figure 7.1 depicts a part of this ontology.\footnote{The complete ontology can be obtained from \url{www.photocay.com/photostock.rdfs}.}

- The \textit{photo} sub-ontology consists of terms coming from the photo domain. It describes things like different kinds of \textit{Light}, various photo \textit{Techniques}, different camera \textit{Settings} etc. The cornerstone of this ontology is the class \textit{Photo} which is linked by its properties with the aforementioned classes. There is also a property called \textit{depicts-Theme} that connects the \textit{Photo} class from the photo ontology with the \textit{Entity} class from the general ontology.

- The \textit{equipment} sub-ontology focuses on the photo hardware, describing and classifying different kinds of lenses, cameras and other related accessories.

- The \textit{general} sub-ontology consists of terms describing all things one can possibly take
a picture of, the most general term being the Entity class. This subpart is typically extracted from general ontologies such as Wordnet [31].

- The ternary sub-ontology serves as a means to describe a story captured on the photograph, where there are two or more actors that perform (either send or receive) an action. For instance, a photo depicting a dog chasing a cat is certainly a different story than that depicting a cat chasing a dog.

Figure 7.1: Photo ontology.

7.2 Existing User Interfaces for Browsing RDFS

In this section we review the two major user interface metaphors frequently used for the visualization and browsing of RDFS-based ontologies: the graph-based approach and the tree-based approach.

We note that there is a number of other complex visualization techniques focusing mainly on class (topic) classification [109; 34]. While these approaches often provide a valuable overview of the domain (sometimes even including the dispersion of instances over the domain), they are less suitable for supporting the generation of path queries as they do not explicitly take into account the class properties. In the rest of this section we focus on approaches that cover both the class and the property related aspects of RDFS.

7.2.1 Graph-based Approach

Since an RDFS based ontology is in fact a graph with labels on edges and nodes, the most straightforward approach is to mimic this structure by a user interface, which itself is a full-
fledged directed labeled graph. This paradigm certainly captures most of the expressive power of RDFS. However, for ontologies with more classes it becomes almost unreadable for the end-user, overwhelming him by its complexity.

The most advanced graph-based user interface we investigated is the one implemented within the KAON project [77]. Except of the standard graph features, it offers several add-ons like gradual exploration (unfolding) of \texttt{rdfs:subClassOf} and \texttt{rdfs:subPropertyOf} hierarchies and the “self-adjusting” algorithm that arranges the appearance of the entire graph by changing the position of nodes and edges each time the user goes deeper in the hierarchy or modifies the graph otherwise. While this feature works fine when applied to a graph consisting of a few nodes, it becomes very slow and not very interactive if one wants to explore a graph with more nodes. Moreover, once the graph is unfolded it is difficult to grasp the hierarchical structure which is “hidden” behind the special edges and
not reflected by the position of the class nodes. This, together with the fact that the graph tends to grow uncontrollably in both dimensions, are the major drawbacks of most graph-based user interfaces. Figure 7.2 shows the KAON graph interface displaying our ontology example, illustrating some of the aforementioned issues.

### 7.2.2 Tree-based Approach

![Tree-based user interface of Protégé-2000.](image-url)

In a tree-like user interface, unlike in the graph metaphor, nodes are organized strictly hierarchically, usually by the property `rdfs:subClassOf` or, when the property centric view is more appropriate, the tree is created following the `rdfs:subPropertyOf` property. This has several positive but also negative implications. On the positive side is the fact that the tree user interface metaphor is very common and its behavior is well understood since the end-users are familiar with browsing trees of different kinds (e.g. deep directory structures, bookmarks, mail folders etc.). Unfolding or browsing a tree can be controlled much better...
both in vertical and horizontal dimension compared to unfolding or browsing a graph. For instance, in the case of a tree-based user interface, each level down the hierarchy moves all ancestors of the current node with a constant distance to the left or to the right. However, unfolding a node in a graph can cause a need to rearrange the whole graph by a variable (hard to predict) expansion in both dimensions.

The drawbacks of the tree approach originate mainly from the fact that an RDFS based ontology is still a graph and there is a discrepancy between a rigid tree structure and a general graph. In other words, it is hard to capture a graph in a tree. For instance, it is difficult in a tree-based user interface to depict things like multiple inheritance or properties that (arbitrarily) associate classes to other classes or to literals. Figure 7.3 shows a tree-based user interface of Protégé-2000 [89], an ontology editor, which we used for creating our ontology example. The Protégé user interface consists of two major parts.

The left part depicts the class hierarchy by a browseable tree below which there is a list of super classes of the currently selected node. The right part mainly consists of a list of properties (Template Slots) together with the (textual) description of their ranges, i.e. class names or primitive types, to which these properties point. While this approach is quite suitable for the design of ontologies, it becomes rather cumbersome for the process of browsing, exploring or familiarizing with an existing ontology, mainly because it is impossible to see at the same time both the domain class together with its context (i.e. the position in the class hierarchy) and the range class together with its context.

7.3 The EROS User Interface

In this section we describe the EROS\textsuperscript{2} interface. Combining the advantages of the graph-based and tree-based approach, one would desire the simplicity of the tree-based approach and at least a part of the expressiveness of the graph based approach. This is exactly what we tried to achieve by the EROS interface depicted in Figure 7.4.

\textsuperscript{2}Explorer for RDFS-based Ontologies
7.3.1 The Class-centric View

The main idea behind this approach is to consider properties as partial mappings that map (some) elements (classes) from the class hierarchy into either other (possibly identical) elements within the same hierarchy, or into a special element called Literal. Note that the set of all elements from the hierarchy serves two purposes: firstly as a (potential) domain of all properties and later as their (potential) range. This double purpose inspired us to actually have two (almost) identical hierarchy trees in our interface, the left tree being the domain (from) tree, and the right tree being the range (to) tree extended with the Literal element. Properties themselves are depicted as arrows connecting the classes from the domain tree with the classes from the range tree. Note that this approach makes it possible to simultaneously display for a certain property both the context of the domain class and the context of the range class. Cases of multiple inheritance are handled the same way as in Protégé-2000, i.e. for a currently selected class in the tree the interface offers a list of its super classes.
7.3.2 Property-centric View and User-defined View

The approach described in the previous section admittedly favors classes over properties as the rdfs:subClassOf property is the key for building the tree hierarchy. However, one may prefer to view the ontology with the "property optics" and desire to explore the tree hierarchy based on the property rdfs:subPropertyOf. The philosophy of the EROS interface can easily accommodate this demand only by imposing that the domain tree is built based on the rdfs:subPropertyOf relationship instead of the rdfs:subClassOf relationship. The rest stays intact. The user now has a view where in the left tree there is a hierarchy of properties connected by their rdfs:domain and rdfs:range properties to the tree on the right which still contains the hierarchy of classes.

In fact, EROS goes even further and allows the user to adapt the interface to his needs by choosing an arbitrary transitive property as a key for building the left tree and another
one for building the right tree. In this case, one can for instance display the resources of his ontology in the left tree-based on the wordnet:hypernym relation and in the right tree on the rdfs:subClassOf property and study the (possibly subtle) differences in these tree hierarchies and their mutual relationships in terms of properties which connect resources from one tree to another. Especially in issues like integrating or aligning ontologies, this ontology mapping is a complicated process and EROS partially offers support for this.

7.3.3 Query Formulation

To facilitate the use of an existing ontology and its instances (annotations), two prerequisites are essential. Firstly, the users have to become familiar with the ontology; here a good and flexible user interface such as EROS is certainly of a great help. Secondly, there must be tools that enable users to express queries over the ontology, execute them, and return resulting instances (resources).

In this section we consider the following user query example over the photo ontology: 

"Retrieve all pictures of Niagara Falls taken from the Falls Avenue in Niagara Falls (Canada) with a lens of focal length 400 mm or more."

The first part of the query denotes the subject (waterfalls) being photographed. The second identifies the position of the photographer. Note the ambiguity of the "Niagara Falls" collocation, first denoting the waterfalls and then being a part of the position description as a town name. Moreover, there is the third part of the query, imposing an additional lens constraint which basically says that we are only interested in those images that provide enough details and a narrow perspective achievable only by using a lens of that focal length.

It is evident that queries of this kind are not likely to be satisfactorily answered by keyword based search engines. By translating this query into a set of keywords and trying a keyword based search engine we either obtained an empty set of results (e.g. Google Image Search ³) or a countless number of irrelevant pictures featuring big photo lenses (e.g. AltaVista Image Search⁴ retrieved over 1.4 million images and the top-ranked ones were indeed mostly showing long lenses and other photo equipment).

³http://www.google.com/imghp
⁴http://www.altavista.com/image/
A full-fledged RDF query language is needed in order to express information needs like these. However, to express the information need in such a query language is often not a trivial task, especially if the user is not familiar with the syntax of the language. To make matters even worse, when it comes to querying RDF(S), there exist several query languages (e.g. [64], [19]) which offer similar query expressive power but require the use of their particular syntax. To illustrate these syntax differences, Figure 7.6 represents the user query in the RQL syntax while Figure 7.7 depicts the same query in the syntax of SeRQL.

We advocate that a large portion of the user query, especially when it involves complex path expressions such as in our example, can be expressed by the actual browsing of the ontology in the user interface. The browsing process can be recursive, consisting of choosing the concepts of interest, examining their properties and following those which are fitting the information need. This approach offers several advantages. First, it prevents
the user to be overwhelmed by the query complexity, as he expresses the query gradually taking one concept at a time. Second, browsing of ontologies is syntax independent and the user interface can translate this process to a syntax of the preferred (or imposed) query language, shielding the user from the burden of learning it.

Of course not all parts of a query are easily expressible by browsing. For instance, to express the filtering conditions typically requires building Boolean predicates and combining them. These predicates consist of variables (defined and bound in the path expressions) compared to literal values or other variables. It is difficult to capture this kind of expressions by browsing the ontology. However, their composition can be aided by the user interface by offering a list of defined (generated) variables and of the Boolean operators.

When in query mode, EROS supports the query creation by browsing, essentially building the query for a chosen query language in a point-and-click manner. EROS allows the user to generate path expressions by selecting a concept in the hierarchy (a variable of a certain type is automatically generated), then selecting a property of the chosen concept that navigates the user to the destination node (which generates again a variable). If the
destination node is not of a literal type, the user can choose another property of that node traversing the graph further (building a chained path expression). As mentioned earlier, the EROS user interface facilitates the property “hopping” in a one-concept-at-a-time fashion. This both shields the user from the complexity of the graph structure and allows him to observe the neighboring concept/property hierarchy. Figure 7.8 depicts the query building mode of EROS for the RQL syntax [64]. Currently, EROS also supports two other RDF query languages namely SeRQL [19] and SNEL [105].

7.4 EROS Summary

This chapter addressed our research question 5 by proposing the EROS user interface.

We have compared two existing approaches for offering browsing support for ontologies: the graph-based and the tree-based approach. We have considered the advantages and disadvantages of both, and derived a new approach that offers the simplicity of the tree-based approach and at least a part of the expressiveness of the graph-based approach.

In the EROS interface the user is able to view the ontology both from the viewpoint of classes and from that of properties (in terms of their domain and range classes). We have shown how this can help in general, but specifically it supports the creation and manipulation of queries on RDF metadata. As an example, we used a photo annotation ontology and a photo retrieval request representing a complex RDF path query. The chosen user interface metaphor proved to be helpful in the process of conceptually “designing” complex RDF path queries by means of ontology browsing. This process is to a large extent syntax independent, and the EROS interface facilitates the actual syntactic implementation of such a query definition process.
Chapter 8

Concluding Remarks

In this chapter we summarize the results of our research and indicate directions for possible future extensions.

8.1 Conclusions

In this dissertation we have addressed several research problems associated with providing support for Web information system design, and for the integration aspects within this process in particular. This section enumerates the research questions from Chapter 1 and provides an overview of the answers that were presented in the different chapters of this dissertation.

Research Question 1

How can we support the design and implementation of Web information systems on the Semantic Web?

We answered this question in Chapter 2 by proposing the Hera framework. This framework encompasses a model-driven methodology for Web information system design and a suite of collaborating software components that execute the resulting design specification. Hera distinguishes two principal design phases: the data collection phase and the presentation generation phase. To overcome the complexity of the overall design task, the methodology further divides these two phases in a sequence of smaller clearly defined design steps. The end-product of each of these steps is a model specification.

Hera distinguishes the following design models:

- The conceptual model which represents an ontology which explicitly captures the conceptualization of the domain of interest and serves as an interface between the data collection phase and the presentation generation phase.
• The integration model that describes a set of mappings between the different sources and the conceptual model. These mappings are needed whenever for a given query the instances that compose the query result need to be retrieved.

• The application model which groups concepts from the conceptual model into meaningful navigational units called slices, and creates a hypermedia structure over them taking into account the adaptation aspects of the WIS design.

• The presentation model that can be seen as a view over the application model, which details the look-and-feel of the presentation.

All of these models are encoded in RDF(S), the acknowledged backbone of the Semantic Web. The choice of RDF(S) as the underlying formalism for all the Hera models effectively promotes reuse and interoperability at different levels of the WIS design process. These models then serve as input for the Hera software suite which by interpreting them realizes the goal of the designer: the implementation of a Web information system.

Research Question 2
How can we specify the integration of semantically heterogeneous RDF sources?

The answer to this question was provided in Chapter 3 where we introduced our integration specification. The main task of the integration model is to provide a semantically unified interface for querying (selected) heterogeneous information sources that populate on request the conceptual model with data. In order to fulfill this task, the proposed integration model vocabulary defines in RDF(S) syntax a number of integration primitives such as ontology path, articulation, and decoration.

The proposed integration model (IM) addresses a number of requirements which were derived from the Hera integration context.

• The IM is expressed in RDF(S)—an open Web standard, which facilitates the reuse of the integrated data as well as of the integration specification itself.

• The IM is expressed at ontology level which follows the data-intensive nature of WIS. Based on the IM, the data instances are reconciled automatically on-the-fly during the query resolution.

• The IM decouples the external sources from the rest of the Hera framework, shielding it from changes in the underlying sources.

• The IM is expressible enough to reconcile heterogeneous ontology paths which are to be linked together. It also offers means for addressing the designation heterogeneity by providing different primitives for the identification of instance data.
The IM includes ranking decorations, in which the designer explicates the otherwise implicit properties of the sources. These decorations are taken into account by the mediator during the query answering process, facilitating a flexible query evaluation which considers the user context.

**Research Question 3**

Given the RDF integration specification, how can it be evaluated and what are the suitable architectures for that purpose?

The answer to this question is provided in Chapter 4. In order to guarantee the freshness of the retrieved data, the Hera integration framework adopts a mediating architecture which implements the on-demand retrieval paradigm in which data is collected on-the-fly from the sources during the query evaluation process. We investigated several types of mediating architectures.

- The basic mediator architecture which establishes transparent access to the underlying repositories containing the RDF data instances that are to be retrieved for the conceptual model.
- The hierarchical mediator architecture, where the mediated sources can be mediators themselves and in turn hide a set of distributed sources.
- The cooperative mediator architecture in which (some) mediators agree to cooperate by offering processing capabilities, especially the join evaluation, to other nodes in the mediating network. This architecture moves us closer towards the P2P paradigm.

One of the main issues in distributed processing of RDF queries is to distribute the sub-paths of a particular query to the appropriate sources. In Chapter 4 we proposed the path index data structure which allows for efficient access to the mapping information form the integration model. Using this data structure we devised an algorithm for efficient distribution of RDF path queries. The main idea is to use the path index to maximize the sub-queries that are being pushed to the underlying repositories; this approach offers several advantages.

- The source repositories contain instance-based indices that allow them to perform local optimization and thus provide for more efficient path query evaluation.
- The underlying repositories process the assigned queries in parallel. The higher the degree of parallelism (i.e. the local query processing), the less work is left for sequential processing at the mediator.
- Pushing down the selective fragments of the query effectively reduces the traffic costs which often contribute largely to the overall execution time.
Research Question 4

How can we optimize distributed RDF query processing?

The answer to this question is provided in chapters 5 and 6. Since it uses an intelligent query routing based on the path index, the query processing described earlier can be considered as the first optimization step in our framework. To also improve the performance of the result assembly, Chapter 5 approaches the RDF result set from the relational point of view, treating it as a number of connected relational tables which need to be joined. In order to speed up the joining process, we deployed a combination of two join optimization heuristics:

- Iterative improvement, a greedy heuristics which randomly generates several initial solutions, taking them as starting points for a walk in the chosen solution space and accepting those steps that yield improvement.
- Simulated annealing, which further improves the results by searching the proximity of the sub-optimal solutions produced by the iterative improvement, while accepting at a certain probability also those moves that result in a solution with a higher cost than the current best solution.

A number of implemented optimization steps (e.g. pushing down the selection operators or reordering the joins) can be perceived as query rewriting. In order to be able to reason about such rewritings in an algebraic manner, Chapter 6 proposed RAL, our RDF algebra which allows to reason about high-level RDF queries in terms of their equivalent query plans, facilitating their rewriting and further optimization.

Research Question 5

How can we help the user with exploration of the application content structure and subsequent query composition?

Chapter 7 addressed our last research question by proposing the EROS user interface for browsing RDF(S)-based ontologies. EROS combines the simplicity of the tree-based approach and the expressiveness of the graph-based approach, facilitating both class-centric and property-centric view over the ontology. EROS also assists in the process of conceptually “designing” complex RDF path queries by means of ontology browsing.

8.2 Possible Future Extensions

In this section we provide pointers to possible future extensions for particular topics of our research.
An interesting extension that is being built into the Hera framework (both in the methodology and in the software suite) is the support for process-aware Web applications. For this purpose the expressive power of RDF(S) does not appear to suffice on its own and needs to be extended. Higher level ontology languages such as OWL [117] are considered in order to be able to describe and reason about more complex systems. With this developments, it became clear that XSLT will no longer provide sufficient means for generating the necessary transformation steps (both in terms of expressivity and performance). Therefore, most of the Hera XSLT components should be ported to a full-fledged programming language like Java.

Another limitation of RDF(S) is that due to its monotonic nature it considers given triplet facts to be valid forever. This approach does not reflect the evolution of (Web) information. For instance, an online conference submission system has the attribute enabled = true only during the submission period, after the submission deadline this attribute changes to false. This kind of changes are often implemented (and hardwired) by the associated application, which many times hinders interoperability among WIS. Hera could benefit from a “time-aware” formalism where the validity of data could be limited by a certain time interval.

Concerning our integration framework, we see a strong need for authoring tools that would help the designer with creating the integration specifications. These tools could also analyze the source ontologies and offer (at least some) initial correlations with our conceptual model. In addition to that, porting our integration model to OWL would provide more expressive power for the designer. This extended expressive power would be obviously required should our conceptual model and the ontologies of the underlying sources be expressed in OWL.

It is clear that more expressive OWL constructs will be also more computationally expensive, and new query processing and optimization techniques will have to be considered. While in our research concerning RDF(S) we focused mainly on database-oriented processing and optimization techniques, the more expressive OWL will require a fair amount of logic-oriented processing and optimizations.

Regarding the distributed query processing and optimization techniques, we remark that our cooperative architecture promotes dynamic load balancing which could be exploited to improve the overall performance even further, especially in systems with nodes of uneven (or volatile) processing speed.

The RDF algebra that we proposed gained recognition among the researchers and practitioners in the field, which resulted in an ongoing effort for its implementation.\(^1\) The future RAL engine targets automatic rewriting and subsequent optimization of queries expressed in several RDF query languages.

The suggestions for future extension of the EROS interface include gradual loading of

\(^1\)http://groups.yahoo.com/group/RDFAlgebra
voluminous ontologies based on a navigation-driven exploration, and presenting (at least
an indication) of the class extent, so that the user can have an estimate of how large the
population of the investigated concept is, even before he issues the actual query.
Appendix A

RDF(S) serialization of the Integration Model

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE rdf:RDF [ 
  <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <!ENTITY articulations 'http://wwwis.win.tue/~Hera/ns/articulations#'>
  <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
  <!ENTITY owl 'http://www.w3.org/2002/07/owl#'> 
]> 
<rdf:RDF xmlns:rdf="&rdf;"
  xmlns:articulations="&articulations;"
  xmlns:rdfs="&rdfs;"
  xmlns:owl="&owl;"> 
  <rdfs:Class rdf:about="&articulations;Articulation"
    rdfs:label="Articulation">
    <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="&articulations;ClassList"
    rdfs:label="ClassList">
    <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="&articulations;Comparator"
    rdfs:label="Comparator">
    <rdfs:subClassOf rdf:resource="&articulations;ProcInstruction"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="&articulations;DataSource"
    rdfs:label="DataSource">
    <rdfs:subClassOf rdf:resource="&articulations;Source"/>
  </rdfs:Class>
  <rdfs:Class rdf:about="&articulations;Decoration"
    rdfs:label="Decoration"> 
```
APPENDIX A. RDF(S) SERIALIZATION OF THE INTEGRATION MODEL

```xml
<rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;Edge"
            rdfs:label="Edge">
    <rdfs:subClassOf rdf:resource="&rdfs;Property"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;From"
            rdfs:label="From">
    <rdfs:subClassOf rdf:resource="&articulations;OntologyPath"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;FromEdge"
            rdfs:label="FromEdge">
    <rdfs:subClassOf rdf:resource="&articulations;Edge"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;FromList"
            rdfs:label="FromList">
    <rdfs:subClassOf rdf:resource="&rdfs;Seq"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;FromNode"
            rdfs:label="FromNode">
    <rdfs:subClassOf rdf:resource="&articulations;PrimaryNode"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;Mediator"
            rdfs:label="Mediator">
    <rdfs:subClassOf rdf:resource="&articulations;Source"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;Mirror"
            rdfs:label="Mirror">
    <rdfs:subClassOf rdf:resource="&articulations;Decoration"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;Node"
            rdfs:label="Node">
    <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;Path"
            rdfs:label="Path">
    <rdfs:subClassOf rdf:resource="&articulations;OntologyPath"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;OntologyPath"
            rdfs:label="OntologyPath">
    <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>
<rdfs:Class rdf:about="&articulations;PrimaryNode"
            rdfs:label="PrimaryNode">
    <rdfs:subClassOf rdf:resource="&articulations;Node"/>
</rdfs:Class>
```
APPENDIX A. RDF(S) SERIALIZATION OF THE INTEGRATION MODEL

<!-- Classes -->
<rdfs:Class rdf:about="&articulations;TransitiveTarget"
  rdfs:label="TransitiveTarget">
  <rdfs:subClassOf rdf:resource="&articulations;To"/>
</rdfs:Class>

<rdfs:Class rdf:about="&articulations;TransitiveToEdge"
  rdfs:label="TransitiveToEdge">
  <rdfs:subClassOf rdf:resource="&articulations;ToEdge"/>
  <rdfs:subClassOf rdf:resource="&owl;TransitiveProperty"/>
</rdfs:Class>

<!-- Properties -->
<rdf:Property rdf:about="&articulations;address"
  rdfs:label="address">
  <rdfs:range rdf:resource="&rdfs;Literal"/>
  <rdfs:domain rdf:resource="&articulations;Source"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;appliesToSource"
  rdfs:label="appliesToSource">
  <rdfs:range rdf:resource="&articulations;Source"/>
  <rdfs:domain rdf:resource="&articulations;Decoration"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;appliesToArt"
  rdfs:label="appliesToArt">
  <rdfs:range rdf:resource="&articulations;Articulation"/>
  <rdfs:domain rdf:resource="&articulations;Decoration"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;backtrack"
  rdfs:label="backtrack">
  <rdfs:range rdf:resource="&articulations;Edge"/>
  <rdfs:domain rdf:resource="&articulations;Node"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;dimension"
  rdfs:label="dimension">
  <rdfs:domain rdf:resource="&articulations;Ranking"/>
  <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;edgeProducedBy"
  rdfs:label="edgeProducedBy">
  <rdfs:domain rdf:resource="&articulations;ToEdge"/>
  <rdfs:range rdf:resource="&articulations;Transformer"/>
  <rdfs:subPropertyOf rdf:resource="&articulations;producedBy"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;ends"
  rdfs:label="ends">
APPENDIX A. RDF(S) SERIALIZATION OF THE INTEGRATION MODEL

```
rdfs:label="idByProperty">
  <rdfs:range rdf:resource="&articulations;Edge"/>
  <rdfs:domain rdf:resource="&articulations;PrimaryNode"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;idByValue"
  rdfs:label="idByValue">
  <rdfs:range rdf:resource="&articulations;Edge"/>
  <rdfs:domain rdf:resource="&articulations;PrimaryNode"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;identicalFor"
  rdfs:label="identicalFor">
  <rdfs:range rdf:resource="&articulations;SubSchema"/>
  <rdfs:domain rdf:resource="&articulations;Articulation"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;mirrorsArticulation"
  rdfs:label="mirrorsArticulation">
  <rdfs:range rdf:resource="&articulations;Articulation"/>
  <rdfs:domain rdf:resource="&articulations;Mirror"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;mirrorsSource"
  rdfs:label="mirrorsSource">
  <rdfs:range rdf:resource="&articulations;Source"/>
  <rdfs:domain rdf:resource="&articulations;Mirror"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;name"
  rdfs:label="name">
  <rdfs:range rdf:resource="&rdfs;Literal"/>
  <rdfs:domain rdf:resource="&articulations;Source"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;nodeProducedBy"
  rdfs:label="nodeProducedBy">
  <rdfs:domain rdf:resource="&articulations;PrimaryNode"/>
  <rdfs:range rdf:resource="&articulations;Transformer"/>
  <rdfs:subPropertyOf rdf:resource="&articulations;producedBy"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;obtainedFromEdge"
  rdfs:label="obtainedFromEdge">
  <rdfs:range rdf:resource="&articulations;FromEdge"/>
  <rdfs:domain rdf:resource="&articulations;ToEdge"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;obtainedFromList"
  rdfs:label="obtainedFromList">
  <rdfs:range rdf:resource="&articulations;FromList"/>
  <rdfs:domain rdf:resource="&articulations;ToEdge"/>
</rdf:Property>
```

<rdf:Property rdf:about="&articulations;obtainedFromNode"
  rdfs:label="obtainedFromNode">
  <rdfs:range rdf:resource="&articulations;FromNode"/>
  <rdfs:domain rdf:resource="&articulations;ToNode"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;producedBy"
  rdfs:label="producedBy">
  <rdfs:range rdf:resource="&articulations;ProcInstruction"/>
  <rdfs:domain rdf:resource="&rdf;Resource"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;rankValue"
  rdfs:label="rankValue">
  <rdfs:range rdf:resource="&rdfs;Literal"/>
  <rdfs:domain rdf:resource="&articulations;Ranking"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;restrictionPath"
  rdfs:label="restrictionPath">
  <rdfs:range rdf:resource="&articulations;Path"/>
  <rdfs:domain rdf:resource="&articulations;Restriction"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;restrictionValue"
  rdfs:label="restrictionValue">
  <rdfs:range rdf:resource="&rdfs;Literal"/>
  <rdfs:domain rdf:resource="&articulations;Restriction"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;source"
  rdfs:label="source">
  <rdfs:domain rdf:resource="&articulations;Edge"/>
  <rdfs:range rdf:resource="&articulations;Node"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;starts"
  rdfs:label="starts">
  <rdfs:range rdf:resource="&articulations;Node"/>
  <rdfs:domain rdf:resource="&articulations;OntologyPath"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;target"
  rdfs:label="target">
  <rdfs:domain rdf:resource="&articulations;Edge"/>
  <rdfs:range rdf:resource="&articulations;Node"/>
</rdf:Property>

<rdf:Property rdf:about="&articulations;toTarget"
  rdfs:label="toTarget">
  <rdfs:domain rdf:resource="&articulations;Articulation"/>
  <rdfs:range rdf:resource="&articulations;To"/>
</rdf:Property>
APPENDIX A. RDF(S) SERIALIZATION OF THE INTEGRATION MODEL

</rdf:RDF>
Appendix B

BNF Grammar of the Supported SeRQL Features

For the complete SeRQL grammar see [20]

Namespace_list ::= "using" "namespace" Namespace ("," Namespace)*
Namespace ::= Prefix "=" Full_uri
Prefix ::= NCName

Query ::= Select_query

Select_query ::= "select" ("distinct")? Projection
("from" Path_expr_list)?
("where" Boolean_query)?

Projection ::= "+"
| Var_or_value ("," Var_or_value)*

Construct_clause ::= "+"
| Path_expr_list

Path_expr_list ::= Path_expr ("," Path_expr)*
Path_expr ::= Path_expr_head (";"? Path_expr_tail)?
Path_expr_head ::= Node Edge Node
Path_expr_tail ::= Edge Node (";"? Path_expr_tail)?

Edge ::= Var
| Uri

Node ::= "{" Var "}"

Boolean_query ::= And_expr ("or" Boolean_query)?
And_expr ::= Boolean_query0 ("and" And_expr)?
Boolean_query0 ::= "(" Boolean_query ")"
| "true"
APPENDIX B. BNF GRAMMAR OF THE SUPPORTED SERQL FEATURES

| "false" | "not" Boolean_query0 | Var_or_value Comp_op Var_or_value | Var_or_value "like" String | "isLiteral" "(" Var ")" | "isResource" "(" Var ")"

Comp_op ::= "=" | "!=" | "<" | "<=" | ">" | ">="

Var_or_value ::= Var | Value

Var ::= NCName

Value ::= Uri | Literal

Uri ::= Full_uri | Local_name

Full_uri ::= "<" (* as defined in http://www.ietf.org/rfc/rfc2396.txt *) ">

Local_name ::= "CM:" NCName

Literal ::= "" (* zero or more (encoded) characters *) ""

String ::= "" (* zero or more (encoded) characters *) ""

NCName ::= (* as defined in http://www.w3.org/TR/REC-xml-names/#NT-NCName *)
Bibliography


[34] Christiaan Fluit, Marta Sabou, and Frank van Harmelen. Supporting user tasks through visualisation of light-weight ontologies.


Samenvatting

Het overweldigende succes van het World Wide Web heeft een verandering teweeg gebracht in het concept van Informatie Systemen (IS), doordat het Web paradigma binnen deze systemen nu zowel als doel platform en als data-bron wordt gebruikt. Het semantische Web levert nieuwe mogelijkheden op die het “oude” World Wide Web niet had. Het brengt echter ook een nieuwe reeks van requirements met zich mee ten opzichte van IS ontwerp. Wij stellen Hera voor, een model gedreven methodologie voor de ondersteuning van Web IS ontwerp, dat zich richt op het integratieproces, data retrieval en presentatie generatie. Alle modellen binnen Hera zijn gebaseerd op RDF(S), de spil-taal van het Semantische Web. Dit maakt de methodologie een geschikte kandidaat voor het ontwerpen van echte Semantisch Web informatiesystemen. Wanneer de inhoud van een dergelijk systeem uit verschillende gegevensbronnen wordt gehaald, vereist de specificatie van hoe de gegevens moeten worden retrieved een geschikt specificatie kader en een suite van tools die de ontworpen specificaties kunnen verwerken en de gegevens kunnen retrieven in reactie op queries van de gebruikers.

Hoofdstuk 2 introduceert het Hera raamwerk. Dit raamwerk omvat een model gedreven methodologie voor het ontwerp van het Web informatiesysteem en de suite van samenwerkende software componenten die een ontwerp specificatie kunnen uitvoeren. Hera onderscheidt twee belangrijkste ontwerpfasen: de fase van de gegevens retrieval en de fase van de presentatie generatie. Om de complexiteit van het gehele ontwerp te versimpelen, verdeelt de methodologie deze twee fasen verder op in een opeenvolging van kleinere helder bepaalde ontwerp stappen. Het eindproduct van elk van deze stappen is een modelspecificatie.

In Hoofdstuk 3 verbijzonderen we het Hera integratie raamwerk door een integratie model voor te stellen samen met zijn RDF(S) representatie. De belangrijkste taak van het integratie model is het scheppen van een semantisch genificeerde interface voor het query-en van (geselecteerde) heterogene informatie bronnen die het conceptuele model bij een gebruikers request vullen met data. Om dit te doen definieert de voorgestelde integratiemodel vocabulaire in een RDF(S) syntax een aantal integratie primitieven zoals een ontologie pad, articulatie en decoratie. Het integratiemodel wordt door de integratie engine gebruikt om een brug te slaan tussen de semantische verschillen tussen semantisch heterogene databronnen en het gespecificeerde Web informatie systeem dat die bronnen gebruikt als zijn data leveranciers.

Hoofdstuk 4 bespreekt verscheidene architecturen van gedistribueerde RDF query engines en gaat in detail in op de gekozen manier van query evaluatie. Een van de belangrijk-
ste kwesties in gedistribueerde evaluatie van RDF queries is het distribueren van sub-paden van een bepaalde query naar de juiste bronnen. Hiervoor stellen we een pad-index data structuur voor die voor een efficiënte toegang zorgen tot de afgebeelde informatie die het integratie model vormt. Met hulp van deze data structuur hebben we een algoritme afgeleid voor de distributie van RDF pad queries.

In hoofdstuk 5 concentreren we ons op de optimaliseringtechnieken in de context van gedistribueerd RDF query evaluatie. Om de query evaluatie in ons gedistribueerde raamwerk the optimaliseren, in het bijzonder de fase waarin de mediator de deelresultaten combineert, mappen we het probleem van query-evaluatie van de RDF pad queries naar het join ordering probleem op het gebied van databases. Om de prestaties van het combineren van deelresultaten te verbeteren, gebruiken wij een combinatie van twee join-optimalisatie heuristieken: iterative improvement en simulated annealing. In dit hoofdstuk beschrijven we ook de implementatie van ons prototype en geven we een performance evaluatie.

In hoofdstuk 6 stellen we RAL voor, n van de eerste algebrasche formalismen voor RDF die vanuit een database perspectief is ontwikkeld. RAL is gebouwd bovenop een gegevensmodel in de vorm van een gerichte gelabelde graaf, waar knopen resources vertegenwoordigen en de kanten properties voorstellen. De RAL operatoren die zijn gedefinieerd over dit model nemen een verzameling input knopen. Deze (expliciete) knopen dienen als uitgangspunt voor de (impliciete) data van het onderliggende model, dat verkregen kan worden door het (recursief) aflopen van de properties van de input knopen. Dus, elke knoop in de verzameling representeert mogelijk een (impliciete) graaf. Net als voor de input, heeft de output van de RAL operatoren ook de vorm van een verzameling van knopen. Dit maakt RAL gesloten onder compositie, wat een belangrijke eigenschap is voor het definiëren van complexe geneste queries.

Hoofdstuk 7 concentreert zich op de gebruikersinterface-aspecten van het ontwerp van WIS en stelt de EROS gebruikersinterface voor, voor het browsen van RDFS ontologieën. EROS combineert de eenvoud van op boom-gebaseerde benaderingen met de expressiviteit van de op graaf-gebaseerde benadering, die zowel een klasse-centrisch als property-centrische beeld geeft van de ontologie. EROS kan ook een hulpmiddel zijn in het proces van het conceptueel “ontwerpen” van complexe RDF pad queries door middel van ontologie-browsing. Dit proces is in hoge mate syntax onafhankelijk, en de interface van de EROS interface helpt het construeren van de eigenlijk syntactisch implementatie in zo’n proces van het definiëren van de query.
Richard Vdovják was born in Liptovský Mikuláš, Slovakia, on February 16, 1974. In 1992 he graduated high school with specialization in programming and in the same year started to study informatics at the University of Zilina. He received his master degree in informatics in 1997 at the same university. Between 1998 and 2000 he followed a Postgraduate Software Technology Program at the Eindhoven University of Technology and obtained a Professional Doctorate in Engineering.

Between 2000 and 2005 Richard Vdovják was a PhD candidate at Eindhoven University of Technology, department of mathematics and computer science. Since 2004 he was also employed as part time assistant professor at the same university.

His research interests include methodologies for engineering Web information systems, information integration, architectures for distributed RDF(S) query processing, RDF(S) query optimization, and RDF(S) visualization.
SIKS Dissertatiereeks

1998

1998-1 Johan van den Akker (CWI)
Degas - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations
within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of
Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The Practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven
Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analysis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgeni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization
2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9 Pieter Jan ‘t Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models,
Views of Packages as Classes
2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent
modeling and simulation language for work practice analysis and design
2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design
====
2002
====
2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis
2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections
2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval
2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic
Environments inhabited by Privacy-concerned Agents
2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based
ontology of the legal domain
2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems
2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble
2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications
2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems
2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications
2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems
2002-15 Rik Eshuis (UT)
   Semantics and Verification of UML Activity Diagrams for Workflow Modelling
2002-16 Pieter van Langen (VU)
   The Anatomy of Design: Foundations, Models and Applications
2002-17 Stefan Manegold (UVA)
   Understanding, Modeling, and Improving Main-Memory Database Performance
====
2003
====
2003-01 Heiner Stuckenschmidt (VU)
   Ontology-Based Information Sharing in Weakly Structured Environments
2003-02 Jan Broersen (VU)
   Modal Action Logics for Reasoning About Reactive Systems
2003-03 Martijn Schuemie (TUD)
   Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy
2003-04 Milan Petkovic (UT)
   Content-Based Video Retrieval Supported by Database Technology
2003-05 Jos Lehmann (UVA)
   Causation in Artificial Intelligence and Law - A modelling approach
2003-06 Boris van Schooten (UT)
   Development and specification of virtual environments
2003-07 Machiel Jansen (UvA)
   Formal Explorations of Knowledge Intensive Tasks
2003-08 Yongping Ran (UM)
   Repair Based Scheduling
2003-09 Rens Kortmann (UM)
   The resolution of visually guided behaviour
2003-10 Andreas Lincke (UvT)
   Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture
2003-11 Simon Keizer (UT)
   Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks
2003-12 Roeland Ordelman (UT)
   Dutch speech recognition in multimedia information retrieval
2003-13 Jeroen Donkers (UM)
   Nosce Hostem - Searching with Opponent Models
2003-14 Stijn Hoppenbrouwers (KUN)
   Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15 Mathijs de Weerdt (TUD)
   Plan Merging in Multi-Agent Systems
2003-16 Menzo Windhouwer (CWI)
   Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17 David Jansen (UT)  
Extensions of Statecharts with Probability, Time, and Stochastic Timing  
2003-18 Levente Kocsis (UM)  
Learning Search Decisions  

====  
2004  
====  
2004-01 Virginia Dignum (UU)  
A Model for Organizational Interaction: Based on Agents, Founded in Logic  
2004-02 Lai Xu (UvT)  
Monitoring Multi-party Contracts for E-business  
2004-03 Perry Groot (VU)  
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving  
2004-04 Chris van Aart (UVA)  
Organizational Principles for Multi-Agent Architectures  
2004-05 Viara Popova (EUR)  
Knowledge discovery and monotonicity  
2004-06 Bart-Jan Hommes (TUD)  
The Evaluation of Business Process Modeling Techniques  
2004-07 Elise Boltjes (UM)  
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes  
2004-08 Joop Verbeek (UM)  
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiediensten: gegevensuitwisseling en digitale expertise  
2004-09 Martin Caminada (VU)  
For the Sake of the Argument; explorations into argument-based reasoning  
2004-10 Suzanne Kabel (UVA)  
Knowledge-rich indexing of learning-objects  
2004-11 Michel Klein (VU)  
Change Management for Distributed Ontologies  
2004-12 The Duy Bui (UT)  
Creating emotions and facial expressions for embodied agents  
2004-13 Wojciech Jamroga (UT)  
Using Multiple Models of Reality: On Agents who Know how to Play  
2004-14 Paul Harrenstein (UU)  
Logic in Conflict. Logical Explorations in Strategic Equilibrium  
2004-15 Arno Knobbe (UU)  
Multi-Relational Data Mining  
2004-16 Federico Divina (VU)  
Hybrid Genetic Relational Search for Inductive Learning  
2004-17 Mark Winands (UM)  
Informed Search in Complex Games  
2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models
2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval
2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams
====
2005
====
2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications
2005-02 Erik van der Werf (UM))
AI techniques for the game of Go
2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language
2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data
2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06 Pieter Spronck (UM)
Adaptive Game AI