I4-D9a

Survey over Existing Query and Transformation Languages Revision 2.0

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D9a/D/PU/a2
Responsible editors:	Tim Furche
Reviewers:	Benedikt Linse and Jakob Henriksson
Contributing participants:	Hannover, Heraklion, Manchester, Munich, Nancy, Venice, webXcerpt
Contributing workpackages:	I4
Contractual date of deliverable:	28 February 2006
Actual submission date:	15 April 2006

Abstract

A widely acknowledged obstacle for realizing the vision of the Semantic Web is the inability of many current Semantic Web approaches to cope with data available in such diverging representation formalisms as XML, RDF, or Topic Maps. A common query language is the first step to allow transparent access to data in any of these formats. To further the understanding of the requirements and approaches proposed for query languages in the conventional as well as the Semantic Web, this report surveys a large number of query languages for accessing XML, RDF, or Topic Maps. This is the first systematic survey to consider query languages from all these areas. From the detailed survey of these query languages, a common classification scheme is derived that is useful for understanding and differentiating languages within and among all three areas.

This revision of deliverable I4-D1 extends that deliverable by a refined version of the presented material that has been published as a chapter in the LNCS tutorial volume for the REWERSE "Reasoning Web" 2005 summer school as well as short outlooks on further refinements of the material for upcoming tutorials and summer school courses.

Keyword List

reasoning, query language, XML, RDF, Topic Maps, OWL, classification, Semantic Web

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Survey over Existing Query and Transformation Languages Revision 2.0

Tim Furche[1], François Bry[2], James Bailey[3], Sebastian Schaffert[4], Benedikt Linse[5] Renzo Orsini[6], Ian Horrocks[7], Michael Krauss[8], and Oliver Bolzer[9]

[1] Institute for Informatics, Ludwig-Maximilians-Universität München
Email: Tim.Furche@ifi.lmu.de

[2] Institute for Informatics, Ludwig-Maximilians-Universität München
Email: Francois.Bry@ifi.lmu.de

[3] Department of Computer Science and Software Engineering, University of Melbourne
Email: http://www.cs.mu.oz.au/ jbailey/

[4] Salzburg Research Forschungsgesellschaft
Email: Sebastian.Schaffert@salzburgresearch.at

[5] Institute for Informatics, Ludwig-Maximilians-Universität München
Email: Benedikt.Linse@ifi.lmu.de

[6] Dipartimento di Informatica, Università Ca' Foscari Venezia
Email: orsini@dsi.unive.it

[7] Department of Computer Science, University of Manchester
Email: horrocks@cs.man.ac.uk

[8] Institute for Informatics, Ludwig-Maximilians-Universität München
Email: Michael.Krauss@ifi.lmu.de

[9] Institute for Informatics, Ludwig-Maximilians-Universität München
Email: Oliver.Bolzer@stud.ifi.lmu.de

15 April 2006

Abstract

A widely acknowledged obstacle for realizing the vision of the Semantic Web is the inability of many current Semantic Web approaches to cope with data available in such diverging representation formalisms as XML, RDF, or Topic Maps. A common query language is the first step to allow transparent access to data in any of these formats. To further the understanding of the requirements and approaches proposed for query languages in the conventional as well as the Semantic Web, this report surveys a large number of query languages for accessing XML, RDF, or Topic Maps. This is the first systematic survey to consider query languages from all these areas. From the detailed survey of these query languages, a common classification scheme is derived that is useful for understanding and differentiating languages within and among all three areas.

This revision of deliverable I4-D1 extends that deliverable by a refined version of the presented material that has been published as a chapter in the LNCS tutorial volume for the REWERSE "Reasoning Web" 2005 summer school as well as short outlooks on further refinements of the material for upcoming tutorials and summer school courses.

Keyword List
reasoning, query language, XML, RDF, Topic Maps, OWL, classification, Semantic Web

Contents

Chapter 1

Introduction

The "Semantic Web" is an endeavor which Tim Berners-Lee, the father of HTML and of HTTP, James Hendler, and Ora Lassila initiated in 2001 with an article in the Scientific American [52]. The "Semantic Web" vision is that of the current Web which consists of (X)HTML and documents in other XML formats being extended with meta-data specifying the meaning of these documents in forms usable by both, human beings and computers:

> The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. [52]

One might see the Semantic Web meta-data added to today's Web as semantic indices similar to encyclopedias. A considerable advantage over conventional encyclopedias printed on paper is that the relationships expressed by Semantic Web meta-data can be followed by computers, very much like hyperlinks can be followed by programs, and be used for drawing conclusion using automated reasoning methods.

> For the Semantic Web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. [52]

A number of formalisms have been proposed in recent years for representing Semantic Web meta-data, e.g., RDF [288], Topic Maps [215], and OWL [35]. Whereas RDF and Topic Maps provide merely a syntax for representing assertions on relationships like "a text is authored by some person", schema or ontology languages such as RDFS [74] and DAML+OIL [210] allow to state properties of the terms used in such assertions, e.g., that no "person" can be a "text". Building upon descriptions of resources and their schemata (as detailed in the architectural road map for the Semantic Web [51]), rules expressed in, e.g., SWRL [208] or RuleML [61], allow the specification of actions to be taken, knowledge to be derived, or constraints to be enforced.

Essential for realizing this vision is the integrated access to all kinds of data represented in any of these representation formalisms or even in standard Web languages such as (X)HTML, SVG, or any other XML format. Considering the large amount and the distributed storage of data available already on the Web, the efficient and convenient access to such data becomes *the* enabling requirement for the Semantic Web vision. It has been recognized that a reasonably high-level, declarative query language is needed for such efficient and convenient access, as it allows to separate the actual data storage from the view of the data a query programmer operates on.

Therefore, the aim of this survey is to provide an overview over the languages considered for each of the major representation formalisms used in the nowadays Web, viz., for XML, RDF, Topic Maps, and OWL. This overview is intended to be valuable for comparing, e.g., RDF query languages among themselves, but to also provide insight on the question, whether a common query language for these representation formalisms is reasonable. Therefore, the following three questions stand at the heart of this survey

- What are the **capabilities of a query language** considered essential for the different areas? Is it possible to identify **common shortcomings** of existing approaches in an area, in particular by means of a **comparison of the issues addressed in that particular area with the other areas investigated**?

- Enabling reasoning, i.e., the ability to derive new knowledge from existing knowledge in a systematic way, is perhaps the most distinguishing feature of the "Semantic Web" vision. Convenient and effective querying in such a setting is likely to require at least some degree of reasoning abilities (e.g., for mediation of data described with differing but convertible vocabularies). Therefore this survey closely investigates, **what reasoning abilities** the query languages offer and **how these reasoning abilities are realized**.

- Indeed, the extent and realization of reasoning abilities proves to be a crucial differential for answering the question, **how to classify the query languages surveyed in this work**. In Section 5.1 a common classification scheme for Web query and transformation languages oriented on their Semantic Web "fitness" is proposed and its usefulness for understanding the differences among the query languages is demonstrated.

To this end, this survey starts in Chapter 2 with a concise introduction into the three representation formalisms considered here, viz. XML, RDF, and Topic Maps. Note, that for most of the discussion OWL is not considered separately, but rather in conjunction with RDF, since there is a number of query languages for RDF that use information represented in (some subset of) OWL for querying. However, in Section 4.5 an approach for querying ontologies represented in OWL is discussed to illustrate the challenges one faces when more powerful ontology languages are considered. Chapter 2 also introduces the scenario used in most of the query language descriptions. A small collection of data about books and their classification is introduced on an abstract level and carefully crafted representations in XML, RDF, and Topic Maps are proposed and discussed.

1.1 Selection of Evaluation Criteria: How to evaluate a Web Query Language?

Based upon this collection of sample data, Chapter 3 proposes (a) an **exhaustive set of evaluation criteria** based on requirements and use cases for Web query languages previously identified in [255, 294, 108, 177, 28, 124, 94] and extended by additional criteria for investigating the Semantic Web "fitness" of the query languages in question. As a means for better illustrating the capabilities of the query languages, the taxonomy of queries proposed in [255] is adapted to the Semantic Web setting and (b) a **small set of queries covering each of the classes in the query taxonomy** is proposed. To compare query languages among different representation formalisms, the queries are presented on a rather abstract level, allowing them to be applied on XML, RDF, or Topic Maps data.

This combined approach has a number of merits compared to previous surveys of Web query languages, cf. [4, 160, 66, 65, 252] (surveys of XML query languages) and [76, 253, 314, 125, 200, 318] (surveys and comparisons of RDF query languages), that have mostly been based on a set of exemplary queries and limited to a small number of evaluation criteria (represented in these queries). However, the crucial aspect of the "user experience" of a language, i.e., how convenient and effective the use of a query language is for solving practical problems, is *not* fully covered by this approach, since this aspect is hard to measure without extensive experimental studies involving users with varying background and expertise. Another noteworthy limitation of this survey is that, for time and space reasons, *not all languages could be covered in the same detail.* Instead, quite a number of languages judged particularly interesting or innovative by the authors of this survey are discussed in more detail, whereas other languages presented highlighting only the most interesting features. However, for all 73 languages the full set of 111 criteria have been evaluated and gathered in tabular form in Appendix B. This limitation is rooted partially in the fact, that there has been a virtual surge of new Web query languages, in particular of Semantic Web query languages (i.e., so far mostly RDF and Topic Maps query languages) in the last two years. Since the beginning of 2004, a dozen new languages have been proposed or existing proposals have been significantly altered. This demonstrates both the high relevance and the relative immaturity of the area of (Semantic) Web query languages. Another area where this survey might be further improved in the future is on the theoretical foundations of the languages considered. Regarding, e.g., formal semantics and data model, only rather general statements are noted here. This is motivated by the lack of consideration of such issues in the majority of the language proposals forming the base of this survey.

Despite these shortcomings the approach taken in this survey also exhibits a number of advantages in contrast to previous surveys of Web query languages:

- considering both query languages for standard and Semantic Web allows a better understanding of what the crucial aspects of Semantic Web query languages might be;

- the far larger number of approaches considered gives the results a broader foundation and applicability;

- the evaluation criteria are, where possible, restricted to easily verifiable properties of the query languages;

- focusing the discussion on a selected set of languages allows more details and a better understanding for that languages.

Finally it should be noted that, as with any such survey, the selection of both criteria and sample queries is certainly subjective and might be biased towards a certain result or language, although the authors of this survey tried carefully to eliminate such bias as far as possible.

1.2 Selection of Surveyed Query Languages: What is a Web query language?

Chapter 4 presents the query languages surveyed in this paper grouped by the underlying representation formalism and, where possible, language "family" (i.e., closely related languages are discussed together to ease the understanding of commonalities and differences).

The selection of languages requires some justification. One basic premise guided this selection: Although the distinction is not always clear, the survey should focus on **languages designed primarily for providing efficient and effective access to data**. This rather narrow basic premise excludes in particular three types of languages that are also sometimes considered query languages or at least related to query languages:

- *Full programming languages and libraries or APIs for accessing XML.* Quite a number of general-purpose programming languages with focus or at least direct support for XML data have been proposed recently, e.g., XMLambda [273], CDuce [40], XDuce [211], Xtatic (`http://www.cis.upenn.edu/~bcpierce/xtatic/`), Scriptol (`http://www.scriptol.com/`), Cω (`http://research.microsoft.com/Comega/`, [272]), and with special focus on Web services and composition XL [165, 166], Scala [289], Water [309]. All of these languages provide some form of specialized data structures for representing and accessing XML data. For existing programming languages, convenient access to XML data can be achieved using some API such as DOM[1], SAX[2], or XmlPull[3] or by means of a language extension, e.g., HaXML [365] for Haskell, XMerL [371] for Erlang, or XJ [201] for Java.

  However, when considering reasoning-aware query languages, the distinction between general-purpose programming languages and query languages becomes blurred, as such query languages are often computationally complete (cf. 5). For the purpose of this survey, a pragmatic approach has been chosen: a language is included, if querying is a core aspect of the language design or the approach to accessing Web data is unique and not covered by other proposals.

- *Evolution and reactivity.* A reactive system allows the specification of the dynamic aspects of a data storage system, i.e., (a) what changes are allowed (b) how to react when a certain event, such as the insertion or deletion of some data occurs. Several proposals for adopting previous approaches such as ECA rules to a Web setting have been published recently. Obviously, there is a close relation between languages for specifying the reactive behavior of a system and those for querying the current state as provided by conventional query languages: reactive languages often employ some query language for evaluating whether (a) the current event matches any of the reactive rules and (b) for conditional rules whether the data is currently in a status matching that condition. However, for this survey only reactive languages that also provide interesting querying abilities are considered. For a survey of reactive languages for the Web, refer to [11].

- *"Rule languages".* Transformations, queries, derivations and reactive behavior can often conveniently expressed in rules. Recently, considerable interest in formalizing the rules guiding business decisions in such a way that they can be (a) understood and possibly even managed without learning a complicated rule syntax, (b) changed rapidly without refactoring existing programs, and (c) used directly to automate or support business decisions such as whether a certain customer may receive a loan or which supplier to use for a certain part. This interest has also triggered the development of numerous, often proprietary languages for "rule engines", i.e., systems that allow the specification and evaluation of such rules often as part of so-called expert systems. Examples for languages often used in this context include Prolog, F-Logic, and various extensions of these languages. In the Web context, the serialization and exchange of rules is particularly interesting as demonstrated by, e.g., the RuleML [61] initiative.

  Again where to draw a line between query languages and general rule languages is not obvious. As in the cases above, in this survey only languages focusing on the efficient and effective querying of data are considered with exceptions for approaches that provide interesting insight for querying.

In the future, it might be interesting to extend the languages covered, e.g., to investigate differences and similarities with respect to requirements, principles, and realization of query languages and reactive, general rule, or programming languages in the Web context.

[1] `http://www.w3.org/DOM/`

[2] `http://www.saxproject.org/`

[3] `http://www.xmlpull.org`

As stated in the introduction, this survey focuses on the language aspect of querying the Web. Therefore, (a) authoring tools such as visual editors are only considered in the context of the query language they are based upon and (b) issues related to storing and indexing Web data are not addressed (for a survey on storage system for RDF refer to [253]).

Despite all these conscious limitations in the kind of languages to be considered, the number of languages still remaining is still surprisingly large. This demonstrates the increasing interest in the area of Web query languages, in particular in RDF and Topic Maps query languages for which the respective standardization bodies have recently started the standardization process for (in the case of RDF, low-level) query languages. However, in neither case the aim to develop a common query language supporting the representation formalisms expected to be at the core of a future Semantic Web has received sufficient priority.

Following the overview of the languages considered, a concise summary of the evaluation results is given in Chapter 5, the details of which are given in Appendix B. From the evaluation results, a classification scheme for Web query languages is derived and briefly discussed by comparing it to previous approaches for classifying (Web) query languages and by demonstrating the ability to provide an insightful view on the languages surveyed here.

The paper is concluded by Chapter 6 with an outlook on possible improvements of this comparison and suggestions on interesting research directions derived from the evaluation.

Two appendices give (a) an overview of different serialization formats for RDF (Appendix A) and (b) the detailed results of the evaluation in tabular format (Appendix B).

Chapter 2

Preliminaries

In this chapter, a concise overview of the three representation formalisms that form the bases for the query languages investigated in this work is given. In particular, a small collection of sample data is described against which queries for assessing the functionality of the query languages considered are evaluated.

The "Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML [...]. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere."[1] XML [71] The previous quote hints at the dual role that XML is currently performing in the Web context: First and foremost, XML provides means for defining the syntax of new languages simplifying specification and deployment considerable, as common issues such as character encoding, markup syntax, linking (ID/IDREF, XLink [142]), mixing markup from different languages, splitting and reassembly of data fragments (XInclude [263]), etc. are handled uniformly at the level of XML.

However, this is not the only reason for using XML: More and more XML, and the underlying semi-structured data model, is recognized as a flexible means for representing, exchanging, and processing heterogeneous data originating from different sources. In this sense, an XML document can be interpreted as a rooted, directed, non-ranked, ordered graph. For many applications one does not consider the various reference or linking mechanisms defined for XML as part of the data model, thus reducing the interpretation of an XML document to a non-ranked, ordered tree. Although this is the data model adopted by the W3C (cf. XML Infoset [134] and XQuery 1.0 and XPath 2.0 data model [159]) and the majority of XML query languages, it is nevertheless recognized that providing means for traversing relations beyond the parent-child and sibling relation conveyed in the tree. Examples for such relations are links established by common keys in ID and IDREF attributes, XLink relations that can also be typed just as in RDF (cf. [135] for a more detailed analysis of the commonalities of RDF and XLink), or application-dependent relations.

As XML has been designed with focus on the first role, some peculiarities such as attributes or the lack of a standardized means, to express that the order of the children of some element is irrelevant and does not be preserved, make processing XML not always as convenient as one might hope, nevertheless XML is and most likely will remain the foundation for most Web application that require the exchange of data and increasingly also for applications where such exchange is not needed.

Basically, an RDF [234, 38] model can be seen as an oriented graph whose nodes are labeled by either URIs, which describe (Web) resources, or literals (elementary data such as strings or numbers), or are unlabeled (the so called anonymous or "blank nodes"). The nodes are connected by arcs, also

[1] http://www.w3.org/XML/

labeled by URIs, which are intended to represent "properties" of nodes (so blank nodes can be used to "aggregate" properties). A common alternative view of such a graph is a set of triples, called "statements", of the form (*Subject*, *Property*, *Object*), where *Subject* and *Object* are graph nodes, and *Property* is an arc. While this model accounts for the use of RDF as a very general description framework of Web resources, properties with special meanings are predefined in the RDF [204] and RDFS [204] specifications [258, 234, 204, 74] to describe, for instance, that a node is the "type" (`rdf:type`)[2] of another one, or is sub-class (`rdfs:subClassOf`) or sub-property (`rdfs:subPropertyOf`), etc. RDFS also defines a number of meta-classes, such as `rdfs:Class`, the class of all classes, or `rdfs:Property`, the class of all properties. The inheritance model exhibits some peculiarities, viz., (a) that resources can be classified in different classes that are not related in the subsumption hierarchy, (b) that the subsumption hierarchy can be cyclic (so that all classes on the cycle are equivalent), (c) that properties are first class objects and, in contrast to most object-oriented subsumption hierarchies, one does not describe which properties can be associated with a class but rather can specify the domain and range of properties. Based upon the information provided by an RDFS schema (or, to use another termed used almost equivalently in this case, ontology) certain inference rules can be specified, e.g., for inferring the transitive closure of the subsumption hierarchy or the type of an untyped resource that has a property associated for which the domain is known. OWL [271, 343, 35] extends the means provided by RDFS for defining the vocabulary used in describing resources.

RDF is designed for the exchange of meta-data represented as resource descriptions in an RDF graph. Therefore, a syntax for serializing and transferring RDF data is required. However, early approaches for an XML syntax of RDF have raised considerable critique, mostly for being overly complex to understand and process. Therefore, a large number of alternative serialization formats have been proposed. Appendix A presents a detailed overview of these serialization formats.

The last representation formalism that forms the basis of some of the query languages investigated in this survey is the ISO Topic Maps standard [215]. Inspired by previous work in the library sciences and on knowledge indexing, the Topic Maps standard [215] defines a data model consisting in a rich set of modeling primitives for representing, structuring, indexing, and relating knowledge. As the previously discussed representation formalisms at the core a topic map is a unranked graph with labeled edges and nodes. The most notable differences to RDF are the richer modeling primitives, that include, e.g., the ability to scope any information provided about a topic and to provide multiple facets of information. Also instead of binary associations, Topic Maps provides *n*-ary associations with roles for distinguishing the members of an association (rather similar to the XLink model for extended links). Topic Maps provides a basic ontology language for specifying a hierarchy of types of topics and associations.

The similarity of Topic Maps and RDF has been recognized and first efforts for integrating the two formalisms are presented in [239, 178].

From the perspective of this survey, all three representation formalisms can be used to represent the sample data discussed below varying mostly in the degree to which standard vocabulary is provided for defining the ontology part of the sample data.

2.1 Collection of Sample Data

For reasons of brevity and consistency, all queries operate on the same data, a collection of information on books. Figure 2.1 shows a graphical representation of an RDF/S graph. For more details see [234, 246, 74]. Note, that some of the RDF statements are represented in a more compact form, e.g., all resources with

[2] Here and in the rest of this paper, common prefixes such as `rdf`, `rdfs`, `owl`, `xsd`, or `xtm` are assumed to be associated with the appropriate namespace.

type `rdfs:Class` are depicted as special nodes instead of explicitly showing the `rdf:type` relation. Also, special arrows are used for the `rdf:type` and `rdfs:subClassOf` relations. Resources are identified with temporary IDs in the fashion of N3 [49], e.g., `_:b1`. Note also that the graphical notation used does not make explicit the connection between the property nodes `translator` and `author` (depicted by blue ellipses) and the instances of this property.

The sample data contains a small ontology using only the subsumption (or "is-a-kind-of") relation `rdfs:subClassOf` and the instance (or "is-a") relation `rdf:type`. This ontology is used to illustrate some of the specific requirements for a Semantic Web query language. We believe, that it is sufficient to show the most interesting issues involved in ontology querying without adding unnecessary complexity. Note, however that both the following discussion and the criteria for the evaluation are for general Web query languages including languages for the standard Web only, i.e., where such ontology information can not be represented in a standardized way but rather using an application dependent vocabulary. Furthermore, several aspects of the underlying data representation formalisms, such as RDF and Topic Maps, influence the desiderata for a Semantic Web query language even when no ontology is involved.

Since all three representation formalisms use the XML Schema simple datatypes defined in [54] for typing scalar data. The book titles and the names of the authors are string literals (either untyped or typed as `xsd:string`). The publication year of a book is typed as Gregorian year (`xsd:gYear`).

The sample data is assumed to be stored at the URL `http://example.org/books#`[3]. Where useful, this URL is associated with the prefix `books`, e.g., for referencing the vocabulary defined in the ontology part of the data.

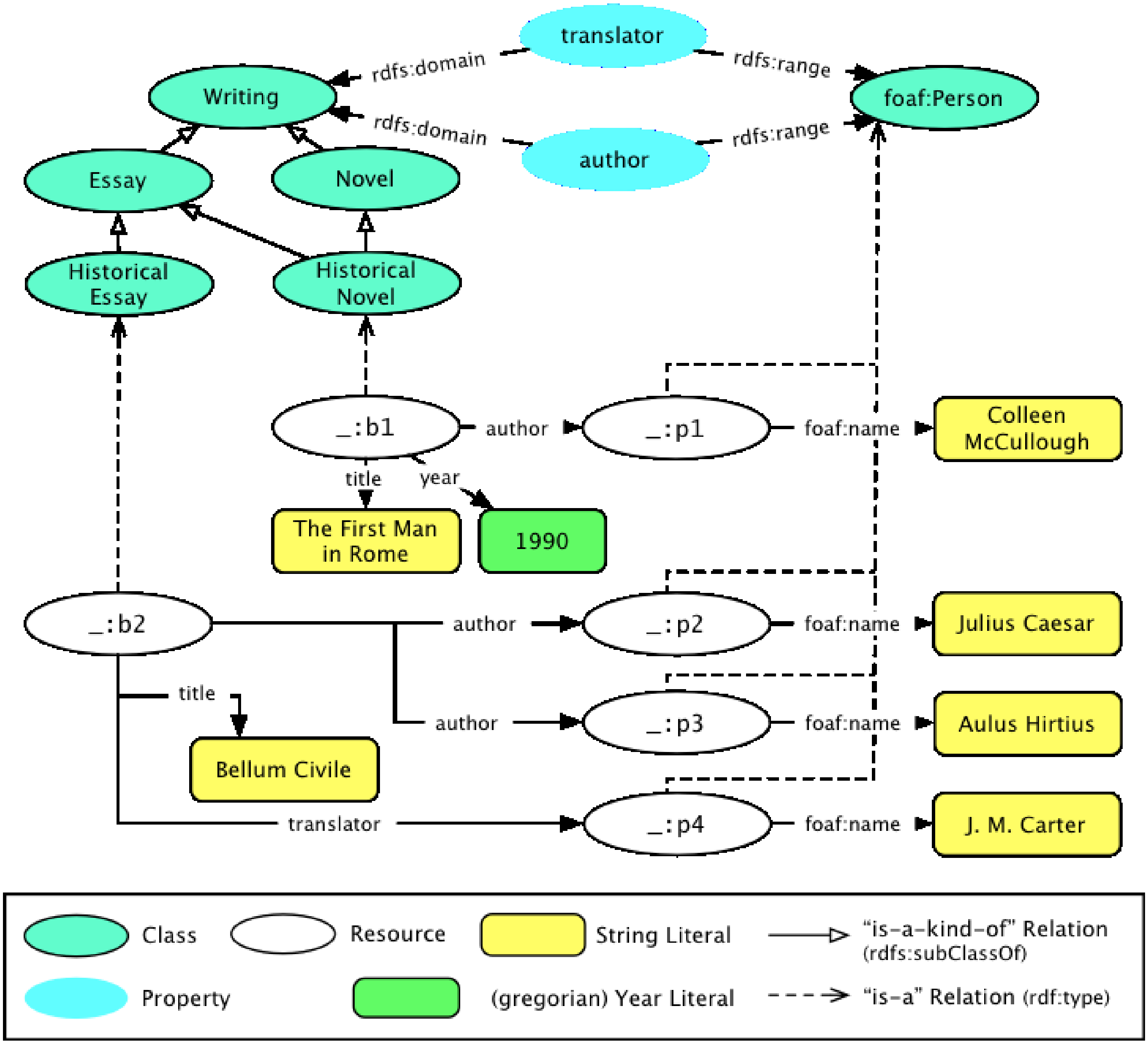
Appropriate **(textual) representation of this data** in the different representation formalisms are chosen as basis for the actual queries. Note, that, since the issue of this survey is not to compare the different representation formalisms, we deliberately chose data and queries that can be handily represented in any of the formalisms.

Sample data in RDF. As the graphical representation is based on the RDF version of the data, this is shown first using the Turtle serialization syntax proposed in [37], a subset of N3 [49] (for more details on the syntax used, cf. Appendix A and the citations).

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .
:Writing a rdfs:Class ;
rdfs:label "Novel" .
:Novel a rdfs:Class ;
rdfs:label "Novel" ;
rdfs:subClassOf :Writing .
:Essay a rdfs:Class ;
rdfs:label "Essay" ;
rdfs:subClassOf :Writing .
:Historical_Essay a rdfs:Class ;
rdfs:label "Historical Essay" ;
rdfs:subClassOf :Essay .
:Historical_Novel a rdfs:Class ;
rdfs:label "Historical Novel" ;
rdfs:subClassOf :Novel ;
rdfs:subClassOf :Essay .
:author a rdfs:Property ;
rdfs:domain :Writing ;
rdfs:range foaf:Person .
:translator a rdfs:Property ;
rdfs:domain :Writing ;
rdfs:range foaf:Person .
```

[3] The URL is chosen in accordance to RFC 2606 [153] on the use of URLs in sample data.

Figure 2.1 Sample Data: Graphical representation of an RDF/S graph



```
_:b1 a :Historical_Novel ;
:title "The First Man in Rome" ;
:year "1990"^^xsd:gYear ;
:author [foaf:name "Colleen McCullough"] .
_:b1 a :Historical_Essay ;
:title "Bellum Civile" ;
:author [foaf:name "Julius Caesar"] ;
:author [foaf:name "Aulus Hirtius"] ;
:translator [foaf:name "J. M. Carter"] .
```

The RDF serialization is, as expected, rather straightforward. Note that both the books and their authors and translators are represented by anonymous nodes (either without identifier or with a temporary identifier indicated by the `_:` prefix.

Sample data in Topic Maps. For the Topic Maps version of the data, the rather compact and readable Linear Topic Maps syntax [176] is used. The subclass-superclass associations are identified using the published subject identifiers defined in XTM [306]. For illustrative purposes the title of a book is represented as an occurrence of that topic. Finally, it is worth mentioning that this representation has been chosen more to demonstrate different features of the query languages surveyed than as a natural expression of the data in Topic Maps. One might, e.g., prefer to use a `publication` association that connects a book with its publisher, the year of publication, and the edition. Also, instead of separate associations for author and translator one could also provide a generic association between persons and books and use appropriate roles for differentiation.

```
/* Association and topic types for subclass-superclass hierarchy */
[superclass-subclass = "Superclass−Subclass Association Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#superclass−subclass" ]
[superclass = "Superclass Role Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#superclass" ]
[subclass = "Subclass Role Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#subclass" ]
/* Topic types */
[Writing = "Writing Topic Type"
@ "http://example.org/books#Writing" ]
[Novel = "Novel Topic Type"
@ "http://example.org/books#Novel" ]
[Essay = "Essay Topic Type"
@ "http://example.org/books#Essay" ]
[Historical_Essay = "Historical Essay Topic Type"
@ "http://example.org/books#Historical_Essay" ]
[Historical_Novel = "Historical Novel Topic Type"
@ "http://example.org/books#Historical_Novel" ]
[year = "Topic Type for a gregorian year following ISO 8601"
@ "http://www.w3.org/2001/XMLSchema#gYear" ]
[Person = "Person Topic Type"
@ "http://xmlns.org/foaf/0.1/Person"]
[Author
@ "http://example.org/books#author" ]
[Translator
@ "http://example.org/books#translator" ]
/* Associations among the topic types */
superclass-subclass(Writing: superclass, Novel: subclass)
superclass-subclass(Writing: superclass, Essay: subclass)
superclass-subclass(Novel: superclass, Historical_Novel: subclass)
superclass-subclass(Essay: superclass, Historical_Essay: subclass)
superclass-subclass(Essay: superclass, Historical_Novel: subclass)
superclass-subclass(Person: superclass, Author: subclass)
superclass-subclass(Person: superclass, Translator: subclass)
/* Occurrence types */
[title = "Occurrence Type for Titles"
@ "http://example.org/books#title" ]
/* Association types */
```

```
[author-for-book = "Association Type associating authors to books"]
[translator-for-book = "Association Type associating translators to books"]
[publication-year-for-book = "Association Type associating translators to books"]
/* Topics, associations, and occurrences */
[p1: Person = "Colleen McCullough"]
[p2: Person = "Julius Caesar"]
[p3: Person = "Aulus Hiritus"]
[p4: Person = "J. M. Carter"]
[b1: Historical_Essay = "Topic representing the book ’The First Man in Rome’"]
author-for-book(b1, p1: author)
publication-year-for-book(b1, y1990)
{b1, title, [[The First Man in Rome]]}
[b2: Historical_Novel = "Topic representing the book ’Bellum Civile’"]
author-for-book(b2, p2: author)
author-for-book(b2, p3: author)
translator-for-book(b2, p4: translator)
{b2, title, [[Bellum Civile]]}
```

Sample data in XML. Here one of many possible XML representations of the sample format is shown. For brevity, the information that authors and translators are persons is not represented. Also, note the use of ID/IDREF links for representing the subsumption data.

```
<bookdata xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<book type="Historical_Novel">
<title>The First Man in Rome</title>
<year type="xsd:gYear">1990</year>
<author>
<name>Colleen McCullough</name>
</author>
</book>
<book type="Historical_Essay">
<title>Bellum Civile</title>
<author>
<name>Julius Caesar</name>
</author>
<author>
<name>Aulus Hirtius</name>
</author>
<translator>
<name>J. M. Carter</name>
</translator>
</book>
<category id="Writing">
<label>Writing</label>
<category id="Novel">
<label>Novel</label>
<category id="Historical_Novel">
<label>Historical Novel</label>
</category>
</category>
<category id="Essay">
<label>Essay</label>
<category id="Historical_Essay">
<label>Historical Essay</label>
</category>
<category idref="Historical_Novel" />
</category>
</category>
</bookdata>
```

Alternatively, an XML serialization of the Topic Maps or RDF data shown above could be used. However, the serializations of both RDF and Topic Maps in XML are rather awkward and would only complicate the understanding of the issues involved in querying XML.

Chapter 3

Evaluation Criteria

This survey is guided by a number of evaluation criteria divided roughly in five areas: ease of use, functionality, semantics, formal properties and implementation, ontology awareness, and reasoning abilities. The evaluation criteria have been derived (1) from a number of relevant use cases and requirements specifications for web query languages [255, 294, 364, 124, 28, 177, 108, 107], (2) from a close look at the capabilities and intended scenarios for deployment for the query languages considered here, and (3) from the design principles and guidelines for the development of a standard and Semantic web query language described in [83].

As discussed in the introduction, this survey is focused on the suitability of the considered approaches for querying the Semantic Web. This emphasis is reflected in many of the following criteria, in particular in the areas of ontology awareness and reasoning abilities.

In this section, the evaluation criteria are introduced, described and an attempt is made to provide a justification for the selection of these evaluation criteria. In Chapter 4, a large number of XML and Semantic Web query languages are closely investigated along the criteria established here. From this investigation, one can easily observe that the proposed criteria allow the identification of several interesting classes of query languages reflecting different approaches, philosophies and requirements for querying the (Semantic) web.

3.1 Ease of Use

It has been previously mentioned that one of the most important aspects when designing a query language (or in fact any language to be used, at least partially, by human beings) is the "feeling" of the language or, in other words, how easy it is to use the language for a given (reasonable) task. Obviously, this is a highly subjective question and hard to measure without empiric studies. Therefore, in Chapter 4 the design philosophy and rational of several query languages are illustrated in more detail and a set of queries is used to give a concrete impression of the languages and their differences and similarities.

Aside of these restrictions, the following aspects of a query language are investigated to give some impression of how convenient the use of the language is:

Syntax. Query languages are often tailored to a specific perception as to who will author queries: whereas expert users usually prefer a *human-readable textual syntax* (C 1.1.), for beginners or casual users even a simple textual syntax might already be too intimidating, however an appropriate *visual syntax* (C 1.2) or a *natural language interface* (C 1.3) can often make a language accessible to such users. Some query languages, e.g., [287], are designed for automatic generation by programs, therefore automatic query manipulation, e.g., by means of an *XML syntax* (C 1.4), is essential. Furthermore, in particular in the semantic web context, the automatic adaptation of queries, e.g., based on ontological data, is an essential issue. In such cases, *meta circularity* (C 1.5) is a desirable language feature, as it allows the adaptation of queries without involving learning and maintaining an additional programming language and environment.

1.1 *Human-readable textual syntax.* Does the query language provide a human-readable textual syntax.

1.2 *Visual syntax.* Is there a graphical editor or visual syntax for the query language?

1.3 *Natural language syntax or interface.* Is it possible to express queries in some kind of (usually restricted) natural language, e.g., in a variant of ACE [170].

1.4 *XML syntax.* Has an XML[1] syntax been specified for the query language.

1.5 *Meta circularity.* Is the query language capable of processing queries written in (at least) one of its syntactic forms.

Extensibility and Modularity. To be able to support a wide range of users with different degree of knowledge and expertise, leads to a set of interesting properties of a query language. Above the use of different syntactical representations of a query for different usage scenarios of the query language has been discussed. Furthermore, many languages provide some mechanism to allow queries to be written in a *modular* (C 1.6) fashion. E.g., views or rules can be provided by more experienced users, if necessary, and used by beginners transparently.

A similar aspect is that of *extensibility* (C 1.7): Different users requires often different functionalities and it is neither desirable, nor in all cases possible to provide all functionality within a single query language or processor. Furthermore, an extensible query language will be better equipped to adapt to emerging use cases in the future.

1.6 *Query modularity.* What kind of constructs for writing modular queries does the language support? Such constructs can be views (cf. [254]), rules (cf. [329, 342]), functions (cf. [57]), etc.

1.7 *Extensibility.* Does the language have a well-defined extension mechanism? Is it possible to detect from within the language what extensions are available in a given environment?

Adherence to Conventions. Both a shallow learning curve and the reuse of previously obtained expertise are supported if a query language uses established conventions where possible. Since this survey considers query languages not only for a single data representation formalism, such as XML, Topic Maps, and RDF, an important criterion for the evaluation is which *data representation formalisms* (C 1.8) are supported by a query language. Although both OWL and RDFS are based on RDF, they are considered separate for this survey, as query languages are likely (and in fact do, cf. OWL-QL) to use the abstract model of OWL rather than query the concrete RDF serialization. Furthermore, both OWL and RDFS introduce constructs (e.g., `rdfs:subClassOf` or `owl:TransitiveProperty`) with a specific semantic that should be supported by a query language.

Related but separate from the issue of the supported data representation formalism is the *data model* (C 1.9) used by the query language. The use of a familiar and appropriate data model will certainly reduce

[1] For this survey, the only syntax considered particularly convenient for automatic processing of queries is XML, as this is the only syntax used for this particular purpose by the surveyed languages.

the time a user requires to become acquaint himself with the query language. Although at first glance one might suspect that all languages for each of the above data representation formalisms use a uniform data model, this is not the case. E.g., some XML query languages consider the data to be strictly hierarchical, i.e., a tree, others offer support for ID/IDREF or similar linking mechanisms, hence use a graph data model. Similarly, some RDF query languages consider RDF data as mere triples, i.e., relational tuples with fixed arity 3 (e.g., [332, 282]), some as arbitrary graph (e.g., [221]), some restrict the graph to be acyclic or rooted (e.g., [302]).

Aside of the data model, also a *syntactical similarity* (C 1.10) with an existing language might be helpful for a novice user. However, often such similarities are merely superficial and can actually impede the understanding of a query language as the intuition from the existing language might not apply or at least not apply in all cases. Therefore, the question, on which *programming paradigm* (C 1.11) a query language is based upon, is often more illustrative of the abilities and general "feeling" of a query language.

Finally, all of the representation formalisms for the Web considered here are based on a graph or tree data model, therefore requiring some *accessor constructs* (C 1.12) that allow the access to specific nodes in the graph or tree based on their position (or relation) to other nodes. Query languages for structured data can be roughly classified by the accessor constructs they provide: *Pattern-based* query languages allow access to several parts of the graph at once specifying the relations among the accessed nodes by tree or graph patterns. *Path-based* query languages use constructs similar to file-system paths to access (usually) a single set of nodes based on any number of relations with other nodes in the graph specified in the path expression. Path-based languages can be further divided in languages that provide only true paths as accessors and languages where it is possible to describe tree-like queries (cf. [121]). Finally, *step-based* query languages provide only constructs for querying the relation of two (sets of) nodes. If it is to be queried whether more than a single relation holds for a certain set of nodes, the multiple relations have to be queried separately and joined via variables. A typical example for a step-based query language is RDQL [332], examples for path-based query languages are XPath [121], RQL [221], and RDFPath [302], Xcerpt [329] is a pattern-based query language.

1.8 *Data representation formalisms.* Which of the data representation formalisms (viz. XML, RDF, RDFS, OWL, Topic Maps) are supported by the query language?

1.9 *Data model.* What data model is used by the query language?

1.10 *Syntactical similarity.* Is there a strong syntactical similarity to other query languages, e.g., SQL, XQuery, or XPath?

1.11 *Programming paradigm.* What programming paradigm is the query language based upon?

1.12 *Accessor constructs.* Is the language based on single steps, paths, or patterns for specifying which nodes in a graph or tree structure to access.

3.2 Functionality

Complementary to the ease of using a query language is its functionality. This survey focuses on three aspects for measuring the functionality provided by a query language: What kind of queries can be expressed in the query language? For which of the concepts of the underlying representation formalism(s) are adequate query constructs provided? Are issues like updates, integrity constraints and active rules considered?

The emphasis on evolution and reactivity might be considered odd, and, indeed, almost none of the languages and systems analyzed in Chapter 4 does provide an update language, let alone means for specifying reactive behavior. Nevertheless, for the Web in general and even more for the Semantic Web, there is a clear need for sophisticated reactive components that allow the fast propagation of and reaction on changes in the (possibly remote) data and other events. Therefore, although support for evolution and reactivity is hardly a very discriminating criterion for the query languages considered in this survey, it is included to illustrate that a strong integration of query languages and reactive behavior is essential for the Semantic Web.

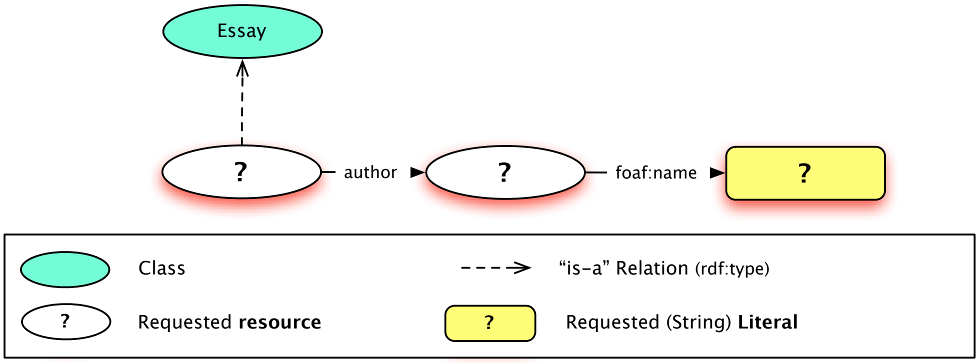
The following discussion details each of these points in order and proposes a set of evaluation criteria that are deemed to be useful for judging what functionality is provided by a query language.

3.2.1 Supported Query Types

For the purpose of this survey five classes of queries have been identified based upon previous work on classifying query languages by the provided functionality, most notably [255, 124]. To illustrate these queries and for later reference, some exemplary queries on the book data from Figure 2.1 are given both in natural language and in an easy-to-understand graphical notation based on the data graph.

- **Selection and Extraction Queries:** (C 2.1.1) The most basic type of query is to ask for some of the information represented in the data, usually based on its content, structure or position within the entire data.

  Query 1. "Select all essays together with their authors and the names of their authors."



  Even such basic queries already raise a number of interesting issues with respect to the capabilities of a query language:

  - *Supported result formats* (C 2.1.2). The query languages considered in this survey differ quite notably in this respect. One of the reasons is the different data representation: Is the data represented in XML, one could, e.g., return a set or sequence of all book elements (possibly already containing the authors and their names) or construct new elements grouping the authors under the related books. However some XML query languages, most notably XPath, do not support the construction of new elements, but always return a set or sequence of the selected elements. For RDF data, one might expect that the statements relating books and authors and the ones associating names to the authors will be returned. However, several query languages for RDF [LIST] do not return triples, but rather a table with one column per variable in the query and one row for each result. Similar considerations apply to Topic Maps.

- *Selection of related information* (C 2.1.3). As in the sample query, one is often not only interested in one piece of information (e.g., the books), but also in related information (e.g., the authors and their names). Usually, it should be possible to obtain the relations from the result, e.g. in the above case one would usually like to know which book relates to which author.

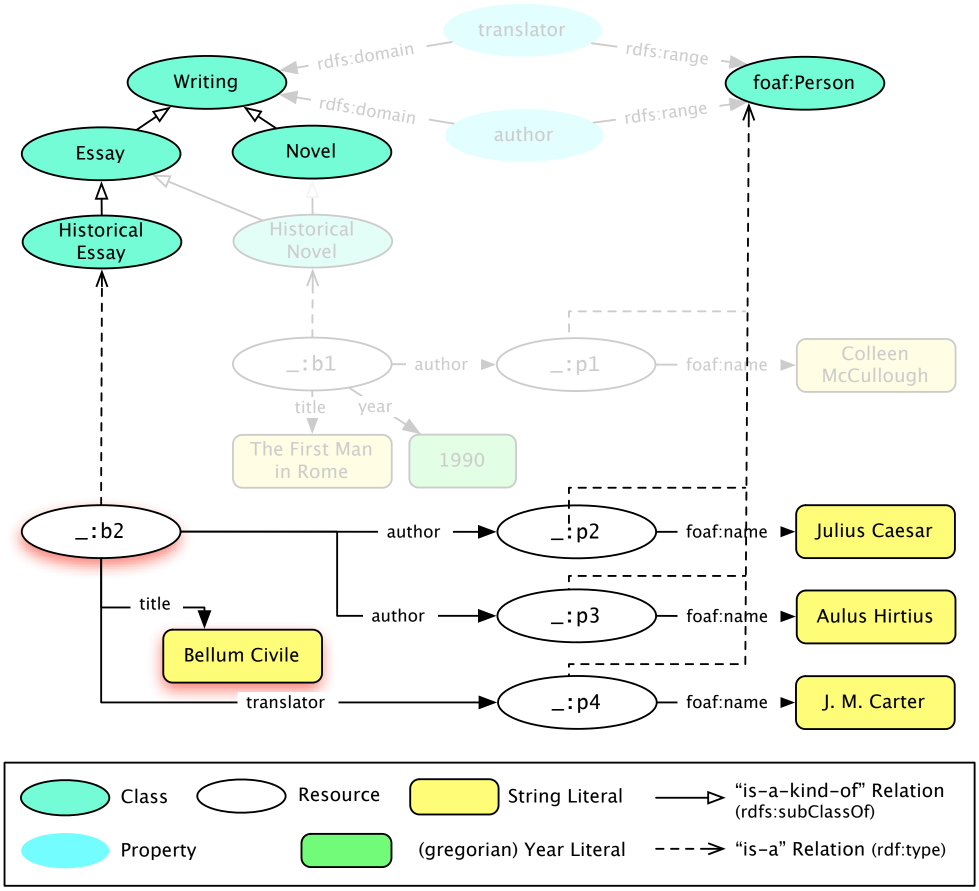
  Again, there are a number of path-based query languages, such as XQL [325], XPath 1.0 [121] or RDFPath [302] that do not provide the ability to select related information.

  But there are also cases where one is simply interested in the information itself without the relations among them. Since representing the relations among the information pieces in the result is expensive, it is desirable to allow both forms of returning result. Whereas XML query languages usually allow the author of a query to make this distinction, most RDF query languages do not provide this possibility.

- Should only books directly classified as "essay" be returned or also ones that are classified in one of its subclasses, e.g., as "historical essay"? In this case, the query actually involves inference, see below.

Another flavor of selection queries that is particularly relevant in a Semantic Web context for collection all information about a particular resource or topic are queries that *extract entire substructures* (C 2.1.4) from the data, e.g., a subgraph of an RDF graph.

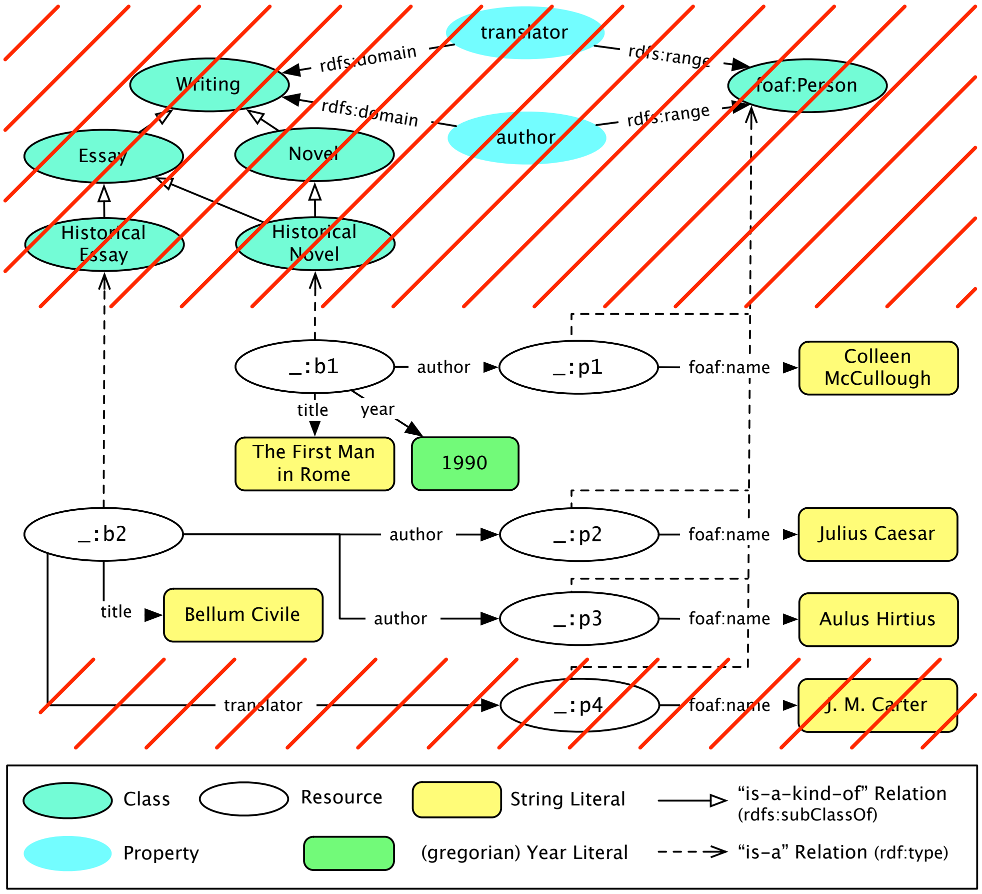
Query 2. "Select everything related to the book with title 'Bellum Civile'."



- **Reduction Queries:** (C 2.1.5) In some cases, instead of specifying precisely what to return, it is easier to specify what should *not* be returned as result. One might, e.g., not be interested in any ontology information or in translators of books for a certain application. The ability to specify what is *not* to be returned, is required, e.g., if the schema of the data to be retrieved is not known in advance, but the schema of the data to be left out is (at least to some extend).

Unless some specific support for reduction queries is offered by a query language, the specification of what should not be returned often requires some form of *negation* (C 2.1.6).

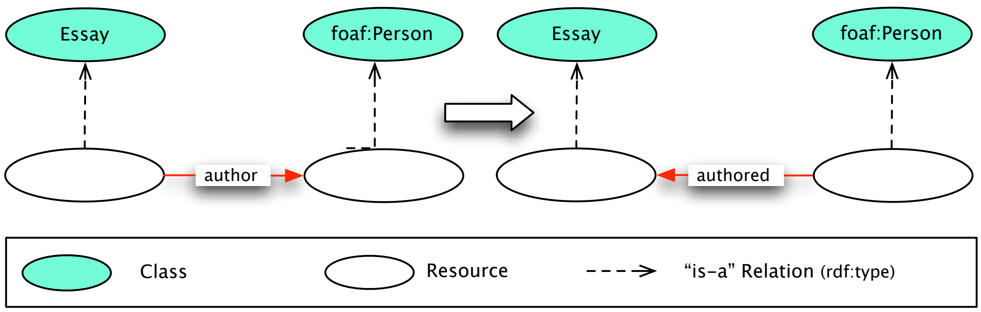
Query 3. "Select everything except all ontology information and any translators."



In the context of the Semantic Web, reduction queries become even more relevant, e.g., for combining information from different sources or for handling trust issues.

- **Restructuring Queries:** (C 2.1.7) Whenever structured data is to be queried, it is imperative to be able to change not only the value of the data but also its structure.

  Query 4. “Invert the relation ‘author’ from a book to an author to ‘authored’.”



In RDF, restructuring of data is used to express statements about statements: A statement that is to be used as subject of another statement is "reified" by assigning an identifier *I* (i.e., an URI) to the statement and transforming the original statement into three statements for specifying subject, predicate and object of the statement.

- **Aggregation Queries:** A simple form of derivation of new knowledge, and hence linked to inference, is the aggregation of data. When structured information is considered, one can not only *aggregate on data values* (C 2.1.8) , cf. Query 5 but also on information about the *structure* (C 2.1.9), as shown in Query 6.

  Query 5. "Return the latest year (the maximum of all years) in which an author with name 'Julius Caesar' published anything (i.e., any 'Writing')."

  Query 6. "Return each of the subclasses of 'Writing' together with the average number of authors per publication of that type."

  Related to aggregation are the concept of *grouping* (C 2.1.10) and *sorting* (C 2.1.11) of the result returned. Note, that grouping and sorting is not meaningful for all of the representation formalisms that form the basis of the query languages discussed here, e.g., in RDF statements do not have any intrinsic order, however sequence container allow the specification of sequences.

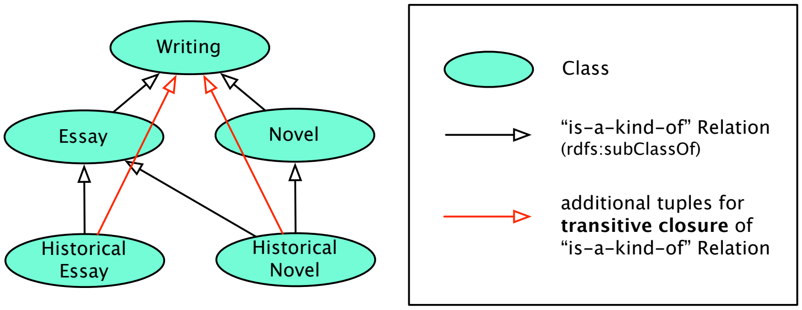
- **Combination and Inference Queries:** Often it is necessary to *combine* (C 2.1.12) existing but not explicitly connected information, e.g., from different sources or represented in varying structures. Many ontologies specify, e.g., which names or identifiers are synonymous, i.e., refer to the same entity.

  Query 7. "Combine all information about a book named 'The Civil War' and authored by 'Julius Caesar' with the information about the book with identifier `bellum_civile`."

  Combination of existing information often allows the *inference* (C 2.1.13) of additional data: if the two books named "Bellum Civile" and "The Civil War" are the same book and "Julius Caesar" is an other of "Bellum Civile" then he is also an other of "The Civil War".

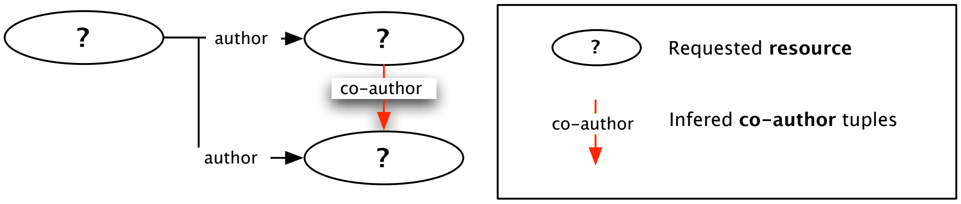
  Another important form of inference queries are queries or views that compute the transitive *closure* of relations such as the `subClassOf` relation used in RDF for defining a subsumption hierarchy.

  Query 8. "Return the transitive closure for `subClassOf` relation."


Writing
Essay
Novel
Historical Essay
Historical Novel
Class
"is-a-kind-of" Relation (rdfs:subClassOf)
additional tuples for **transitive closure** of "is-a-kind-of" Relation

Not all inference queries are combination queries, as the following example illustrates, where a new relation is (intensionally) defined based on existing data about books:

Query 9. “Return the co-author relation between two persons that stand in author relation with the same book.”



Whereas some query languages provide special closure operators for specifying which relations are transitive, others limit queries on the transitive closure to a fixed set of relations, e.g., the `subClassOf` relation from RDFS. Finally, some query languages provide a *general recursion* mechanism, that allows among others to query the transitive closure of arbitrary relations, even those defined intensionally as above.

2.1.1 *Selection queries*. Does the query language support selection queries?

2.1.2 *Supported result formats*. In what serialization or representation formalism can the result be returned? Possible values are, e.g., input subset (only a subset of the input can be returned), arbitrary XML, restricted XML (with an explanation of the restriction), one of the RDF serializations (cf. Appendix A, one of the Topic Map serializations, table (usually with one column for each variable to be included in the result and one row for each match), set or sequence (for query languages lacking the ability to select related information).

2.1.3 *Selection of related information*. Is it possible to return related information within the same query?

2.1.4 *Substructure extraction*. Are the means provided to select an entire substructure (e.g., a subtree or subgraph), in particular if the extent of the substructure is not known in advance?

2.1.5 *Reduction queries*. Does the query language support reduction queries?

2.1.6 *Negation*. Is it possible to express negation in the query language, e.g., to test the non-existence of data or to specify reduction queries.

2.1.7 *Restructuring queries*. Is it possible to rearrange the structure of the input or to create an entirely new structure?

2.1.8 *Aggregation on data values*. Can data values be aggregated in the query language?

2.1.9 *Aggregation on structure*. Is it possible to aggregate over the structure of the date, e.g., to determine the maximum number of authors for a book?

2.1.10 *Grouping*. Does the query language support grouping?

2.1.11 *Sorting*. Does the query language support sorting?

2.1.12 *Combination queries*. Is it possible to combine information that is not directly connected by the structure, e.g., by means of a join over some identifier?

2.1.13 *Inference queries*. What means for inference queries are provided by the query language? Possible values are, e.g., closure on predefined relations, closure on arbitrary structural relations (e.g., parent-child and sibling relation in XML or any property in an RDF graph), general recursion.

3.2.2 Adequacy

Considering the often still immature and evolving nature of the current representation formalisms for the Web and the often frustrating lake of a common and formal understanding of the underlying data models, it is not surprising that there is a great variability among the supported features and interpretations of the underlying representation formalisms among the query languages discussed here: Exemplary issues are, e.g., whether to consider an XML document with ID/IDREF links as graph or tree data or whether to support the more controversial features in RDF, e.g., reification and containers.

Therefore, we consider for each of the representation formalisms separately a number of criteria to given an impression of what a query languages supports. These criteria are based on observing where the surveyed query languages differ and not meant to be a comprehensive list of features of the representation formalisms.

- **XML:** The XML data model (as defined in [134]) is an ordered tree and therefore provides two basic structural relations among nodes (representing elements in the document) in a tree: the *parent-child* (C 2.2.1) and the *sibling* (C 2.2.2) relation. XML query languages should support queries involving either relation and their *closures* (C 2.2.3), i.e., ancestor-descendant and preceding-following relations. Note, that XML languages that employ paths for accessing nodes often allow such relations to be queried in both directions, e.g., from the parent to the children as well as from a child to its parent. However, in [297] it has been shown that a restriction to queries where the relations are followed in "forward" direction (w.r.t. the order in which nodes are visited) is reasonable. A third relation particular to the XML data model is provided via *ID/IDREF links* (C 2.2.4). If these relations are handled transparently, an XML document actually has to be represented as a graph (cf. C 1.8).

  Support for the intrinsic order of an XML document often goes beyond merely providing constructs for querying the sibling relation: many XML query languages allow to *access nodes by their position* (C 2.2.5) within some sequence of related nodes, e.g., the second title of each book. Some query languages also allow *unordered matching* (C 2.2.6), where the order among siblings is ignored. Finally, all XML query languages *preserve the order* (C 2.2.7) when returning some part of the input unless the result is reordered specifically.

  One particular important aspect when considering XML data is that the data often lacks a fixed schema or the schema allows for a certain amount of flexibility. Therefore any XML query language should be able to specify both *partial queries* (C 2.2.8), i.e., queries where only some constraints on the data are specified and the existence of additional nodes in the data does not affect the matching, and *total queries* (C 2.2.9), i.e., queries that only match if there is no more data than specified in the query. Also often desirable is the ability to specify *optional* (C 2.2.10) parts of a query: if there is some data matching an optional sub-query it will be returned, but if no data matches the query other related items are still returned: E.g., one might want to select all books together with their translators but still return the books if there is no translator for it. Such optional sub-queries resemble outer or inner joins in relational databases.

  Another important distinction among XML query languages is the support for *construction of new elements* (C 2.2.11). It is prerequisite for supporting restructuring queries.

  Finally, there are some issues concerning the alignment of an XML query language with the emerging standards defined for XML: support for *XML Schema* [156,357,54] (C 2.2.12)—more details on typing are discussed below in Section 3.6.1—, support for *namespaces* (C 2.2.13), support for advanced linking using *XLink* [142] *and XPointer* [189,188,140,141] (C 2.2.14), and construction of compound documents specified with *XInclude* [263] (C 2.2.15).

- **RDF:** Beyond simple triple statements that associate two resources (called subject and object) via a certain property (or predicate), the RDF data model has some peculiarities that require special attention when discussing a query language for RDF. Even more than in the case of XML, RDF has been design under the assumption that in a global Semantic Web fixed schema information is often unattainable. Therefore, all properties in RDF are *optional* (C 2.2.10) and *multi-valued* (C 2.3.1), e.g., a book described in RDF can have any number of authors including none. A query language should provide suitable constructs for optional sub-queries as in the case of XML. Furthermore, properties are, as any resource, identified by an URI and can therefore be the subject of other statements. In this case, it is required that a query language is able to *query (the identifiers) of properties* (C 2.3.2), and not only those of subjects and objects. Another aspect of this are containers and collections in RDF: *containers* (C 2.3.3) provide the means for expression sets, sequences and alternatives of resources, e.g., to express that a committee has voted in a certain way without implying that every single member actually voted in that way. Within sequences *access by position* (C 2.2.5) is often useful, e.g., to obtain the first in the sequence of chapters for a book. Alternatives also have to be handled differently from sets or sequences since it can only be derived that at least one of the objects is actually related to the subject (by the given predicate). *Collections* (C 2.3.4), introduced only recently during the revision [38] of the original RDF proposal [246], differ from containers in their semantics: collections can be "closed" in the sense that no further elements can be included in the collection (for example when consulting additional descriptions on the same resource). This is not the case for RDF containers. Note, that both containers and collections can be reduced to triples (i.e., binary relations), however requiring considerable effort by the user. Therefore, specific support for these constructs takes a considerable burden from the query programmer.

  A similar consideration applies to one of the more controversial abilities of RDF, viz. to be able to express statements about statements via *reification* (C 2.3.5). As stated above, A statement that is to be used as subject of another statement is "reified" by assigning an identifier *I* (i.e., an URI) to the statement and transforming the original statement into three statements for specifying subject, predicate and object of the statement. An RDF query language should provide at least support for the transparent access to reified statements, i.e., the query programmer should not have to specify whether a statement is reified or not.

  E.g., for the attribution of what the source of a statement is, a statement is often associated with additional information about its context. Although such context informations are often useful and desirable [194, 232], this can only be realized in RDF using reification. Therefore, the Jena Toolkit [190], one of the syntactical forms for RDF [327] discussed in Appendix A and a recent query language [315] all use the concept of *"quads"* (C 2.3.6), i.e., triples enriched with an additional context or source information. This allows not only a more compact formulation of queries but can also be used for improved storage of such statements (cf. [148]).

  As mentioned in Chapter 2 and discussed in further detail in Appendix A, there is a plethora of different serialization formats for RDF. Therefore, it is interesting to note, which *serializations* (C 2.3.7) can be used for input and for the result of a query.

  Issues related to typing and classification of resources are considered when discussing RDFS, OWL and the ontology awareness

- **Topic Maps:** Topic Maps [215], as discussed, are an ISO standard for representing "information about the structure of information resources". As such, they have several commonalities with RDF, but are separated in a clearer focus on the application area in which they are intended to be used. In [28] and [177] use cases and requirements for a upcoming Topic Map query language have been detailed. A large number of these requirements are covered by criteria discussed in other sections of this survey. However, from previous approaches for the conceptualization of information, Topic Maps inherit a number of rich modeling concepts that should be supported by a Topic Maps query language: In contrast to the binary relations of RDF, Topic Maps provide *n-ary associations* (C 2.4.1) with *labeled roles* (C 2.4.2) identifying the associated data. As in RDF, association can be further described by other associations, requiring that the query language supports the *querying of association names* (C 2.3.2). Also similar to RDF and XML, which associations can be used for a certain topic type is not fixed, thus *optional* sub-queries (C 2.2.10) are often helpful. There is a number of *predefined associations* (C 2.4.3), e.g., for defining a sub-class hierarchy similar to RDFS, some of them with special semantics (e.g., transitivity). Also query languages should support some of the more advanced concepts of Topic Maps, such as *scopes* (C 2.4.4), that allow to limit the validity of an association and *facets* (C 2.4.5) used to create filters.

Within Topic Maps, subject identity can be established by the use of subject indicators (also known subject descriptors). A query language should be able to connect information on the same topic provided, e.g., from different sources, by *querying these subject indicators* (C 2.4.6).

As with RDF, there is a number of different proposals for *serializing Topic Maps* (C 2.4.7), e.g., XTM [306], LTM [176], and AsTMa= [27], and it should be noted which of these are supported by a Topic Maps query language.

One of the most prominent differences between Topic Maps and RDF is that essentially in Topic Maps all "statements" are already reified, i.e., for each association there is a node representing that association (for more details, [349]. Therefore, no special treatment of reified "statements" is necessary in Topic Maps.

Aside of the concrete aspects of the different representation formalisms, there is a number of features for a query language that are provided by languages specifically designed to be used in a Web context. Such query languages have to deal with inherent heterogeneity in a global distributed system. *Querying multiple data sources* (C 2.5.1) and providing *multiple (often different) answers to multiple outputs* (C 2.5.2) is just the basis to such querying. The query language should also provide means to reduce the amount of data being transfered between Web sites, e.g. by *limiting the result size* (C 2.5.3), support for *reduction queries* (C 2.1.5) limiting the answers to what is actually of interest to the user and by supporting for easy *distribution of sub-queries* (C 2.5.4) based on the data sources accessed, e.g., by clearly identifying which parts of a query deal with what data. This relates to the *composability of queries* (C 2.5.5): Often the result of one query is to be further refined, possibly at another Web site. To maximize interoperability, the query language should support different *output serializations* (C 2.5.6) and provide some mechanism for the *user to specify* (C 2.5.7) which serialization to choose.

Since data sources are heterogeneous (e.g., w.r.t. the level of structure in the data provided) and may, in general, contain erroneous or contradicting data, means for *vague or approximate query answering* (C 2.6.1) and *relevance ranking* (C 2.6.2) of query answers are often necessary to deal with Web data sources. Approximation should be possible not only as in classical information retrieval on the text content of the data but also on its structure. To some extent, this is already covered by the above discussion, e.g., through constructs expressing optional sub-queries that allow for certain variation in the data to be matched. However, some query languages might provide additional constructs for *approximate structure matching* (C 2.6.3). Due to the varying level of structure provided by the data sources, some kind of text processing can often not be avoided. Therefore, Web query languages might also include classical information retrieval features as proposed for querying XML data, e.g., in [171] and in the recent extensions of XQuery and XPath [13]. Such features include *full-text queries* (C 2.6.4) such as single word-search, phrase search, ordered multi-word search, and proximity search based on word distance or structure, and *word normalization* (C 2.6.5), e.g., stemming, stop-word handling, suffix and prefix removal, or the use of thesauri, dictionaries or taxonomies.

Data sources are heterogeneous not only with respect to the data representation, but also in their communication abilities. E.g., some data sources might provide data rather slowly, therefore making it undesirable to wait with processing that data until all necessary data has actually arrived. In such a case, an implementation providing *progressive processing* (C 4.12) of the data can often provide answers long before all data has arrived. The dual case are data sources that provide data so rapidly that conventional techniques for parsing and storing that data (either in memory or in a database) are infeasible. A *streamed* (C 4.11) implementation of a query language evaluates queries directly against the stream of incoming data without requiring expensive data structures to be built.

2.2.1 XML *Parent-child relation*. Does the query language provide access to the parent-child relation?

2.2.2 XML *Sibling relation*. Is it possible to query the siblings of a node?

2.2.3 XML *Closure relation*. Are there means to query the closure of the two base relations parent-child and sibling.

2.2.4 XML *ID/IDREF*. Does the query language support the explicit or transparent dereferencing of ID/IDREF (or similar) linking mechanisms? Possible values for this criterion are: transparent (i.e., links are automatically resolved and can be queries like parent-child relations), explicit (i.e., there is some specific construct to be used for following ID/IDREF links), indirectly (i.e., it is possible to dereference ID/IDREF links but only by querying the actual XML attributes—no specific constructs are provided) and none.

2.2.5 XML,RDF *Access by position*. Is it possible to access nodes in the structure based upon their absolute or relative position within the structure? In RDF, this only applies for (sequence) containers.

2.2.6 XML *Unordered matching*. Is it possible to specify several children for the same node in the XML structure such that the order among the children is not relevant for finding a match?

2.2.7 XML *Order-preserving result*. If the result of a query is a part of the input, is it possible to preserve the order?

2.2.8 XML *Partial queries*. Is it possible to specify only partial constraints on the data to be matched by a query, e.g., allowing for additional children of a node to exist?

2.2.9 XML *Total queries*. Is it possible to specify that a match must fulfill exactly the constraint given in the query, disallowing the existence of additional data.

2.2.10 XML,RDF,TM *Optionality*. Is it possible to express that certain sub-queries are optional, i.e., that their result should be included in the total result if they match anything, but if they do not match, the remainder of the query may still yield result?

2.2.11 XML *Construction*. Does the query language provide means for constructing new elements and attributes?

2.2.12 XML *XML Schema*. Does the query language make use of XML Schema [156, 357, 54] or a similar schema language such as RELAX NG [122, 123], e.g., for typed queries or type checking of programs or validation of result?

2.2.13 XML *Namespaces*. Does the query language support XML namespaces as defined in [69, 70]?

2.2.14 XML *XLink and XPointer.* Is there some specific support for extended linking mechanisms as provided by XLink [142] and XPointer [189, 188, 140, 141]?

2.2.15 XML *XInclude.* Does the query language allow the construction of compound documents using, e.g., XInclude [263]?

2.3.1 RDF *Multi-valued properties.* Is it possible to query and return properties with multiple values?

2.3.2 RDF,TM *Querying property or association identifiers..* Is it possible to query the identifiers of properties or associations, e.g., for finding properties of properties?

2.3.3 RDF *Containers.* Does the query language have some provisions for container support, e.g., for construction of containers, querying a sequence?

2.3.4 RDF *Collections.* Are there some specific constructs for querying and returning collections (defined in [204])?

2.3.5 RDF *Reification.* Is there some specific support for reification in the query language? Such support can be, e.g., the transparent querying of reified statements or an easy notion for querying and constructing reified statements.

2.3.6 RDF *Quads.* Does the query language offer specific support for context or source information associated with a statement (often represented as “quads” [327])?

2.3.7 RDF *RDF serializations.* Which of the different RDF serializations discussed in Appendix A are supported by the query language and its implementations?

2.4.1 TM *Querying n-ary associations.* Is it possible to query relations of arbitrary arity?

2.4.2 TM *Labeled roles.* Is it possible to use roles for querying and construction and to query the role labels?

2.4.3 TM *Predefined associations.* Does the query language support predefined associations such as subclass-superclass relationship and their special semantics?

2.4.4 TM *Scopes.* Is their explicit support for scopes in the query language?

2.4.5 TM *Facets.* Is their explicit support for facets in the query language?

2.4.6 TM *Subject indicators.* Are (published) subject indicators supported for connecting information from different data sources?

2.4.7 TM *Topic Maps serializations.* Which of the Topic Maps serialization formats are supported by the query language and its implementations?

2.5.1 *Multiple data sources.* Is it possible to query multiple data sources?

2.5.2 *Multiple outputs.* Is it possible to generate different answers for sending to multiple Web sites?

2.5.3 *Limiting result size.* Is there some mechanism for limiting the size of the result retrieved (possibly in conjunction with an SQL-like offset construct for paged answer retrieval)?

2.5.4 *Sub-query distribution.* Does the query language provide convenient means for identifying reasonable sub-queries for distribution to the data sources, if these provide such a capability?

2.5.5 *Composability of queries.* Is it possible to compose queries?

2.5.6 *Output serializations.* Which output serializations are supported by the query language? Here, only general values such as RDF, XML or Topic Maps are given. The different serializations for RDF and Topic Maps are investigated as criterion C 2.34 and C 2.41.

2.5.7 *User-specified serializations.* If there are different serializations provided, can the user choose the serialization (either implicitly by constructing the appropriate serialization directly or by selecting explicitly the desired output serialization)?

2.6.1 *Approximate query answering.* Is there a provision in the query language for answering vague or approximate queries?

2.6.2 *Relevance ranking.* Does the query language automatically rank answers by their relevance or provide some means for explicit ranking or scoring of answers?

2.6.3 *Approximate structure matching.* Does the query language support approximate matching only on the content or also on the structure of the data?

2.6.4 *Full-text queries.* Are there means for processing full-text content of the structural data to support, e.g., word or phrase queries?

2.6.5 *Word normalization.* If there is some support for full-text and word queries, is it possible to apply some word normalization such as stemming or normalization based on thesauri before matching?

3.2.3 Evolution and Reactivity

As discussed above, evolution and reactivity are important concepts linked to a query language that become even more important in the context of the Semantic Web. Three aspects of evolution and reactivity are particularly related to query languages (for a more detailed survey, cf. [11]):

2.7.1 *Update language.* Does the language provide updates or has a related update language been defined?

2.7.2 *Integrity constraints.* Is there some mechanism provided to define and enforce rules that specify restrictions on how the data can be changed?

2.7.3 *Event handling.* Is it possible to specify that certain actions should be performed if an event, such as an update, occurs?

3.3 Semantics

A clear and *well-understood formal semantics* (C 3.1) enables not only a better understanding of the workings of a query language, but also a certain independence from the actual implementation as queries written for one implementation of the language should be usable with any other implementation that follows the semantics. Furthermore, a formal semantics also proved to be very fertile for the development of various kinds of implementation-independent optimizations ranging from source-to-source transformations to operator reordering in logical query plans. These optimizations have, in contrast to optimizations on the level of the physical query plan, the advantage that they usually are usually not specific to a single implementations, but rather can be applied to a number of implementations (e.g., with similar characteristics). Such optimizations often benefit from two characteristics of a formal semantics:

compositionality (C 3.2) and *referential transparency* (C 3.3), as these enable “local” optimizations where the context in which a sub-query occurs does not have to considered. Finally, for any kind of reliable reasoning (e.g., where also proof traces are to be delivered), a formal and well-understood semantics is indispensable.

As a reference for implementations and as basis for cost estimations used in query optimization, an *operational semantics* (C 3.4) with a rigid mapping into the formal semantics is often desirable. Such an operational semantics can, e.g., be provided by means of an *abstract machine* (C 3.5) together with a translation from the formal semantics into instructions of that abstract machine.

3.1 *Formal semantics*. Has a (well-understood) formal semantics been established for the query language?

3.2 *Compositionality*. Is the semantics of the language compositional, i.e., defined in such a way that the semantics of a compound construct is based on the semantics of its parts?

3.3 *Referential transparency*. Is the semantics of the query language constructs referentially transparent, i.e., not depending on the context?

3.4 *Operational semantics*. Has an operational semantics (together with a mapping from the formal into the operational semantics) been defined?

3.5 *Abstract machine*. Has an abstract machine for the query language been defined?

3.4 Formal Properties and Implementation

Aside of a formal semantics, also the status and properties of implementations for a query language is worth noting. This gives some impression on the capabilities of the query language that are also related to formal properties such as complexity and completeness.

In the past, in particular for relational databases, two properties have often be touted to be distinguished features of query languages in contrast to general purpose programming languages: *declarativness* (C 4.1), i.e., that a query language describes what should be result and not how that result can be obtained, and no *computational (Turing) completeness* (C 4.2).

[326] defines a declarative language as a language where each operation is declarative, i.e., “*independent* (does not depend on any execution state outside of itself), *stateless* (has no internal execution state that is remembered between calls), and *deterministic* (always gives the same results when given the same arguments)” [326]. This definition is adopted for the purpose of this survey.

Historically the lack of computational completeness found in many relational query languages such SQL has been perceived as an advantage both for query writing and query execution and optimization. Where computational completeness has been required either stored procedures written in some general-purpose programming language or the embedding of a query language into a host programming language have been used. However, in the last decade it has been recognized (e.g., during the development of the latest SQL standard—sometimes collectively referred to as “SQL3”) that computational completeness actually provides benefits in many cases: Embedding in a host language and stored procedures have both proven to be troublesome with respect to interoperability, performance (in particular for embedding in a host language) and authoring of queries. In this survey, we therefore also note whether a query language is *computationally complete* or not.

In particular for expressive (or even computationally complete) query languages, the issue of scalability w.r.t. efficiency is crucial: It should be possible to characterize *interesting sub-languages* (C 4.3) that can be evaluated with different complexity, such that simple and frequent queries can be evaluated rather quickly, but queries using more expressive features of the query language might actually take longer. In particular, it should be noted, whether a *polynomial core* (C 4.4) has been identified for the query language, i.e., a sub-language such that all queries written in that language can be evaluated in polynomial (combined) complexity. Often the evaluation strategies for such sub-languages differ, therefore an efficient *automatic classification* (C 4.5) of queries in terms of their computational cost is highly desirable.

Additional to the formal properties discussed, this survey also includes a brief overview over the **implementations** provided for a query language. The following issues are investigated: *number* (C 4.6) of different implementations, *status* (C 4.7) of the implementation (e.g., prototype, internal production use, external production use), and *support* (C 4.8) by major database or Web technology vendors such as Oracle, IBM, HP, Microsoft, or large open-source projects such as Apache.

As discussed above, different kinds of implementations of a query language are very desirable in a Web context with varying application requirements and capabilities of data sources. Therefore we note for each query language whether it has been implementation on top of a *database* (C 4.9) (i.e., for querying persistent data where updates are rare and queries are frequent), in an *in-memory query processor* (C 4.10) (i.e., where both query and data are used once only), and in a *streamed* (C 4.11) fashion (i.e., where persistent or continuous queries are evaluated against volatile data). In each of these cases, answers might be provided once all data has been processed or in a *progressive* (C 4.12) manner, i.e., as soon as possible.

4.1 *Declarativeness.* Can the query language be considered declarative following the definition from [326]?

4.2 *Computational completeness.* Is the query language computationally complete?

4.3 *Interesting sub-languages.* Have interesting sub-languages been defined, e.g., with different complexity characteristics?

4.4 *Polynomial core.* Has a sub-language with polynomial complexity been proposed?

4.5 *Automatic classification.* If there are sub-languages with different complexity characteristics, has a method been proposed to automatically classify queries?

4.6 *Number of implementations.* How many different implementation have been developed for the query language?

4.7 *Status of implementations.* What is the status of these implementations?

4.8 *Support.* Is there support for the query language by major database or Web technology vendors or large open-source projects.

4.9 *Database implementation.* Has the query language been implemented on top of a database?

4.10 *In-memory implementation.* Has an in-memory processor for the query language been developed?

4.11 *Streamed implementation.* Has a streamed implementation been provided or considered?

4.12 *Progressive implementation.* Has a progressive implementation been provided or considered?

3.5 Reasoning

Reasoning or the ability to derive additional data based upon the actual data stored in the database has been an important ability of deductive or logic databases [361, 106] and reflected in query languages such as Datalog. Since reasoning is to be considered a defining element in the Semantic Web vision, it is suitable to ask what kind of reasoning mechanisms query languages to be used in this context provide.

The first step to reasoning support is the ability to *intentionally specify* (C 5.1) data, e.g., by means of rules, views or functions. Although this has already been considered under C 1.6 ('query modularity'), it will be noted here again to illustrate where simple derivations of new data are possible. To specify such a derivation, boolean operators or equivalent constructs are often used: *conjunction* (C 5.2, realized, e.g., by set intersection), *disjunction* (C 5.3, realized, e.g., by set union), *negation* (C 5.4, realized, e.g., by set difference), and *quantification* (C 5.5, realized, e.g., by relational division). Also being able to specify optional sub-queries (as discussed in C 2.2.10 ('optionality') can ease the specification of derivations.

Based upon derivations as described in the previous paragraph, more powerful reasoning abilities can be provided. Recursion, in particular, allows the specification of complex derivations such as, e.g., the transitive closure of relations or associations. Several forms of recursions are provided by query languages: *general recursion* (C 5.6) where rules, views, functions or similar intentional data specifications can be recursive, *structural recursion* (C 5.7) where some means (e.g., a special operator for computing the transitive closure of a relation) for recursion along the structure of the data is possible, and transitive closure only over some *predefined relations with special semantics* (C 5.8).

Together with how to specify inference, it is also necessary to note what *inference theory* (C 5.9) is used by a query language (in other words, what kind of reasoning is actually provided). Since the Web is constantly evolving and the appropriate reasoning may differ between different domains, some query languages provide *theory extensibility* (C 5.10), i.e., provide a well-defined interface for adding new reasoners that either implement a different inference theory or provide additional reasoning abilities for specific domains (e.g., ontology reasoning by a description logics reasoner or temporal reasoning).

5.1 *Intensional data specification.* Does the query language support the intensional specification of data, e.g., by means of views?

5.2 *Support for conjunctions.* Does the query language support the use of conjunctions or similar operations?

5.3 *Support for disjunctions.* Does the query language support the use of disjunctions or similar operations?

5.4 *Support for negation.* Does the query language support the use of negations or similar operations?

5.5 *Support for quantification.* Does the query language support the use of quantification or similar operations?

5.6 *General recursion.* Is it possible to use general recursion in the query language?

5.7 *Structural recursion.* Is it possible to traverse the structure of the data recursively (e.g., by means of a recursive relation `descendant` in XPath [121])?

5.8 *Closure on predefined relations.* Are there some predefined relations that can be traversed recursively for accessing the transitive closure (e.g., `rdfs:subClassOf` in RQL [221])?

5.9 *Inference theory.* What inference theory is used by the query language, if any?

5.10 *Theory extensibility*. Is it possible to use different inference theories or to add reasoning abilities for specific domains?

3.6 Ontology Awareness

A query language to be used in a Semantic Web context should be able to incorporate ontologies: Ontologies can be used to query across data sources with different vocabularies for describing the data by providing a mediation between these vocabularies. They can also help to improve the recall of a query by extending the actual queried terms with semantic information (e.g., related terms, contextual information).

To leverage ontologies for querying, the ontologies and the semantic relations described within have to be queried. Although recent ontology languages such as RDFS [74] and OWL [271, 343, 35, 304] are building upon RDF (in that they provide a defined set of terms for RDF that allows the definition of new vocabularies), merely support for RDF is not sufficient to be able to access the knowledge contained in the ontologies appropriately. Rather, a query language has to be aware of the semantics of the terms provided by RDFS or OWL, e.g., the "is-a" relation (`subClassOf` relation in RDFS). Thus, the query language can use the ontology to derive new knowledge about the described instances, e.g., through property propagation. Therefore, the querying the *data described by the ontology and the ontology* (C 6.1.1) itself should be possible within the same language. The following issues are investigated to classify the level of support for ontology languages (for this purpose, only RDFS and OWL are considered, since these are the considered by some of the surveyed query languages):

- **RDFS:** RDFS [74] provides only a small set of terms for describing vocabularies in RDF. The semantics of these terms is defined in [204]. Some of these terms have specific properties, e.g., `rdfs:subClassOf` and `rdfs:subPropertyOf` are both transitive, others allow a limited form of reasoning, e.g., if a class C is the range (i.e., the set of possible values) of some property p, written as an RDF triple $(p, \texttt{rdfs:range}, C)$, and (x, p, y) (some x has the property p with value y, then one can infer that y is an element of class C, i.e., $(y, \texttt{rdf:type}, C)$.

  The query languages surveyed here show different *support for RDFS* (C 6.2.1): Most languages treat RDFS terms as any other RDF term, i.e., without special consideration. Some languages support `querying the transitive closure` (C 6.2.2) of `rdfs:subClassOf` and `rdfs:subPropertyOf`, of which some provide transparent support, others require the explicit specification of the transitive closure (e.g., by means of a recursive rule, view or function or using a special closure operator). Only a small number of languages also use *RDFS for typing* (C 6.2.3), thus providing special constructs, e.g., to query the extent of a class, and static type checking.

- **OWL:** Based on previous ontology languages such as DAML+OIL, the recently specified Web Ontology Language (OWL) [271, 343, 35, 304] is starting to see wide-spread acceptance in academia and also for certain industrial applications. OWL supports a much larger set of terms for defining and constraining vocabularies. Since the support for OWL is still rather limited (with the notable exception of OWL-QL [163]), this survey only addresses some general issues related to *support for OWL* (C 6.3.1): support for *special property classes* (C 6.3.2) such as `owl:TransitiveProperty` or `owl:SymmetricProperty` (e.g., when querying properties that are classified as `owl:TransitiveProperty` the transitive closure of the property should be used), *information propagation* (C 6.3.3) for classes in the subsumption hierarchy (defined by `rdfs:subClassOf`) and for class equivalence, intersection, etc., and *information propagation* (C 6.3.4) for individuals (e.g., when using `owl:sameAs`).

  As in the case of RDFS, also OWL can be used for *typing* (C 6.3.5) the described data.

6.1.1 *Querying both ontology data and instance data.* Does the query language support querying the ontology data together with the data described by the ontology?

6.2.1 RDFS *Support for RDFS.* Does the query language support RDFS?

6.2.2 RDFS *Transitive closure for subsumption hierarchy.* Is it possible to query the transitive closure for the RDFS subsumption hierarchy (created using `rdfs:subClassOf` and `rdfs:subPropertyOf`)?

6.2.3 RDFS *Use of RDFS for typing.* Is the type system of the query language (at least partially based) on RDFS and the `rdf:type` relation?

6.3.1 OWL *Support for OWL.* Does the query language support OWL?

6.3.2 OWL *Special property classes.* Are the classifications of properties w.r.t., e.g., transitivity and symmetry, used for querying?

6.3.3 OWL *Information propagation for classes.* Is information about classes (based on the subsumption hierarchy and class equivalence etc.) propagated for querying?

6.3.4 OWL *Information propagation for individuals.* Is information about individuals propagated for querying?

6.3.5 OWL *Use of OWL for typing.* Is the type system of the query language (at least partially based) on OWL and the `rdf:type` relation?

3.6.1 Type system

Related to ontologies and schema languages is the issue of **typing** in Web query languages. For the purpose of this survey, we have already introduced two criteria related to typing, viz. the use of ontologies specified in RDFS or OWL for typing information. For XML, type information can obviously also be provided by XML schema languages such as XML Schema [156,357,54] or RELAX NG [122,123]. This survey only gives a brief overview of typing related questions in query languages (following [98]):

6.4.1 *Typing.* Have typing issues at all been considered for the query language?

6.4.2 *Static vs. dynamic typing.* Does the query language support static or dynamic typing.

6.4.3 *Explicit vs. implicit typing.* Is data typed by explicit type declarations or implicitly (e.g., by type inference as in the statically-typed Haskell or by type inspection as in the dynamically-typed Smalltalk).

6.4.4 *Type inference.* Does the query language provide type inference, e.g., to avoid explicit type declarations.

6.4.5 *Type coercion.* Is it possible to change the type of an expression, either automatically or using, e.g., a cast operator?

6.4.6 *Support for XML Schema simple data types.* Both RDF and XML Schema use the same set of "simple" data types defined in [54]. Therefore, this criterion notes whether these simple data types are supported by the query language?

Based upon these 111 evaluation criteria, the following section presents an overview of the evaluation followed by a short description of the evaluated languages.

Chapter 4

Query Languages for the Web: An Overview

4.1 XML Query Languages

In this chapter, a very brief overview of some XML query languages is provided. For reasons of space and time, only some of the vast number of XML query languages proposed are considered. In particular, no visual query are left out for they are hard to evaluate by the evaluation criteria discussed here. Recently, there has been a number of proposals for combining full-text querying capabilities provided in information retrieval systems with structured access to XML data. Again, for reasons of space and time, these languages have not been considered.

4.2 Textual XML Query Languages

4.2.1 Navigational Languages

4.2.1.1 Lorel

Lorel [6] is a query language originally designed for semistructured data (more specifically, the language OEM [303, 180]) that was later adapted to XML data. Its syntax strongly resembles SQL and OQL, but it is capable of navigating graph structures in a path-like fashion.

Query 10 (Lorel). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
select xml(results:
(select xml(result:
(select X.author, X.title
from bib.book X
where X.@year > 1991))
))
```

Lorel is a navigational language with rule-like `select-from-where` queries. It is capable of querying several documents and evaluating joins, but order is not considered when querying, as semistructured data is usually always unordered. Multiple data items can be retrieved by assembling several path selections. Although the result of a query is a set of object identifiers (OIDs), XML elements can be constructed using expressions of the form `xml(tagname:subexpression)`. As it descends from OQL, it is capable of grouping and supports aggregations. Queries can be nested, in which case query and construction are not separated. Lorel supports a basic type system but is not aware of XML schema information and does not use type inference.

4.2.1.2 XPath

The *XML Path Language* (XPath) [121] is a W3C recommendation for a selection language whose primary purpose is to address parts of an XML document. Since it lacks construction and reassembling aspects, it cannot be considered a full-fledged query language and is thus called a *selection language* in this thesis. Many other XML query languages build upon XPath, most prominently XQuery and XSLT.

XPath expressions specify navigation steps within the data tree represented by a document, relative to a so-called context node (which is initially the root node of the document). An XPath expression consists of several location steps separated by `/`, each specifying *how to reach* a node relative to the previous node's position. XPath can thus be considered as a *navigational* language.

Query 11 (XPath). The following XPath expression selects titles of books with author `Dan Suciu` and year attribute with a value greater than 1991:

```
/bib/book[@year > 1991][author = "Dan Suciu"]/title
```

Literally, this expression reads: from the root, go to `bib` elements, from there to `book` elements for which holds that the attribute `year` is greater than 1991 and that contain an *author* element as child node with a text value of `Dan Suciu`, and select all `title` child nodes.

XPath differentiates axes like `child`, `descendant`, `parent`, `ancestor` or `sibling`. These axes can be classified into *forward axes*, which contain all such navigations that only move forward in the document tree and *backward axes*, which contain all such navigations that only move backward in the document tree [297]. Since XPath allows both forward and backward axes, evaluation can be very complex and a node might be visited several times during a selection. However, [297] shows that arbitrary XPath expressions without variables can be rewritten into equivalent XPath expressions containing only forward axes.

Forward XPath expressions resemble positional pattern, as they no longer specify arbitrary navigations through the document tree, with the minor exception that a pattern usually does not allow to match the same node in the document by two different nodes of the pattern:

Query 12 (XPath Forward). In the forward XPath expression

```
/a[child::b]/*[att="some value"]
```

the selection $*$ and the `child::b` might match the same node in the database:

```
<a>
<b att="some value"/>
</a>
```

4.2.1.3 XQL

XQL [325, 323] is a variant of XPath that has been proposed and implemented by Microsoft and others in lieu of XPath becoming a completed recommendation. It differs from XPath only in minor points that are not relevant to this comparison.

4.2.1.4 XSLT

XSLT, the *Extensible Stylesheet Language* [120], is a language for transforming XML documents. Originally intended as a powerful style-sheet language, it is often considered as a query language as well, and the existence and development of two independent W3C XML query languages is often criticized. As it was the first available query language for XML, XSLT is very widespread and understood by many programmers. A multitude of implementations exist (e.g. as part of a standard library for XML processing in Java).

An XSLT style-sheet is composed of one or more transformation rules (called *templates*) that recursively operate on a single input document. Transformation rules are guarded by patterns, which are expressed in terms of XPath expressions. The first rule whose pattern matches is evaluated, all other rules are ignored. In contrast to most other query languages, XSLT uses an XML-only syntax:

Query 13 (XSL). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/bib">
<results>
<xsl:apply-templates/>
</results>
</xsl:template>
<xsl:template match="book[@year > 1991]">
<result>
<xsl:apply-templates select="title"/>
<xsl:apply-templates select="author"/>
</result>
</xsl:template>
<xsl:template match="title|author">
<xsl:copy-of select="."/>
</xsl:template>
</xsl:stylesheet>
```

This stylesheet is interpreted as follows:

- try to match the root node with the templates in the style-sheets (only first template matches)
- create a `<results>` element and within it try to recursively apply the templates to all child nodes of `<bib>`
- for each child node, if the year is greater than 1991, create a `<result>` element and recursively apply the rules to the `<title>` and `author` children of the context node
- for each `<title>` and `<author>` element, copy the complete input to the result.

Like XQuery, XSLT is based on the selection language XPath and thus a navigational language. Since a template always operates on a single node, neither retrieval of multiple data items nor joins are directly supported. It is, however, possible to assemble several XPath expressions within a construction pattern. XSLT always operates on a single input document and is not capable of retrieving data from more than one resource. It partly supports ordered/unordered data by using XPath. XSLT allows to construct new data and grouping, but aggregations are limited to those supported by XPath. Although XSLT is a rule-based language, it does not really support the separation of construction and querying, as each rule only applies to a single node. Rules may be called explicitly to form sub-queries, and such calls may be recursive (i.e. XSLT is Turing complete [229]). XSLT is an untyped language that is unaware of any available schema or type information.

4.2.1.5 XQuery/Quilt

XQuery [57] can be considered the "state-of-the-art" XML Query language as it is the current W3C recommendation for XML querying and therefore very widespread. XQuery has several predecessors, of which it resembles most the language *Quilt* [110], but influences from other languages (like XQL and XML-QL) are reflected in many constructs.

XQuery queries consist of so called FLWOR (FOR-LET-WHERE-ORDER BY-RETURN) expressions and use XPath (described above) for the selection of data items. `FOR` and `LET` serve to bind variables to values selected by XPath expressions. Whereas `LET` binds a variable to a set of data items, `FOR` iterates over the different data items in a set. The `WHERE` part may be used to specify conditions for the selected data items. `ORDER BY` has only been introduced recently and allows to order the results in a certain sequence. `RETURN` marks the beginning of a result pattern, which may contain additional XPath selections. XQuery expressions are enclosed in curly braces and embedded in the construction pattern.

Query 14 (XQuery). Select all authors and titles of books written after 1991:

```
<results>
{
FOR $book in document("bib.xml")//book
WHERE $book/@year > 1992
RETURN <result>
{ $book/title }
{ $book/author }
</result>
}
</results>
```

This query iteratively binds the variable `$book` to all book elements occurring in the document in the `FOR` part. The `WHERE` part ensures that only such books are selected that have an attribute year with a value larger than 1991. The `RETURN` part gives a construction pattern that itself again contains subqueries for selecting the title and authors of the book.

Being based on XPath, XQuery classifies as a navigational query language. Multiple data items can be selected only by using multiple XPath expressions. XQuery supports arbitrary nesting of queries as well as the definition of external functions that may contain frequently used subqueries. XQuery is aware of both schema information and basic types and some implementations support static type checking (e.g. Galax[1]). XQuery is not rule-based and heavily mixes querying and construction, making more complex queries difficult to read.

4.2.1.6 FXT

The language *fxt* [45], the *functional XML transformer*, is a transformation language which is similar to XSLT in that it uses the same kind of pattern-guarded rules to recurse over the input document. However, *fxt* aims at optimal performance and thus puts certain limits on patterns and path expressions. Most importantly, *fxt* neither supports explicit calling of rules (thus no recursion other than over the document tree) nor iteration constructs like XSLT's `for-each`. However, allows to perform auxiliary computations by embedding *SML* expressions in transformation rules.

Query 15 (fxt). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
<fxt:spec>
<fxt:pat>/bib</fxt:pat>
```

[1] http://db.bell-labs.com/galax/

```
<results>
<fxt:apply/>
</result>
<fxt:pat>//book</fxt:pat>
<fxt:if test='fromString(Vector2String(getAttribute (String2Vector "year") current))
> 1991'>
<result>
<fxt:apply select="/author"/>
<fxt:apply select="/title"/>
</result>
</fxt:if>
<fxt:pat>//author</fxt:pat>
<fxt:copyContent/>
<fxt:pat>//title</fxt:pat>
<fxt:copyContent/>
</fxt:spec>
```

fxt is based on the path language *fxgrep* and thus classifies as a navigational language. Interestingly, it allows to select (at most) two data items at once by using so-called *binary patterns*. The restricted path language allows neither joins nor to differentiate between ordered and unordered queries, and a transformation always operates on a single input document. *fxt* allows to construct new elements, but aggregations and grouping can only be performed by reverting to the underlying functional language SML. *fxt* is rule-based but neither allows rule chaining nor separates construction from querying. It supports basic types but does not take advantage of schema information and performs no type inference outside SML expressions.

4.2.1.7 XPathLog

LoPiX [269] is an implementation of the XML querying and data manipulation language XPathLog [270]. XPathLog aims at integrating F-Logic with path-based selection in XML documents. Queries in XPathLog are specified as conjunctions of path expressions in which variables may occur at multiple positions. It is thus possible to select several nodes in a single selection step. Further queries may refer to variable bindings. Two elements can be merged by so-called *object fusion*, which appears to be a mechanism similar to feature unification. This mechanism can also be used to group elements.

Instead of the transformation approach taken by most other query languages, where an input document is transformed into a new output document, XPathLog allows to update existing documents. Updates are specified by Prolog-like rules where the right-hand part consists of a query and the left-hand part specifies how to modify the document.

4.2.1.8 CXQuery

CXQuery, the *Constraint XML Query Language* [114], is an effort to create a declarative query language with support for schema information. It uses rules similar to Datalog and can use XPath to navigate the document tree as well as term-based patterns, but apparently, no deep structures are possible (as in Datalog). Construction is specified by term structures in the head of a rule. Again, nesting does not appear to be possible.

4.2.2 Positional Languages

4.2.2.1 XML-QL

XML-QL [143] is a positional, rule-based query language for XML and was designed at AT&T Labs. It uses an XML-based pattern language where variables may occur at arbitrary positions. An XML-QL query consists of a single `CONSTRUCT-WHERE` rule which may be divided into several subqueries.

Query 16 (XML-QL). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
WHERE
<bib>
$book
</> IN "bib.xml"
CONSTRUCT <results>
WHERE <book year=$y>
<title>$t</>
<author>$a</>
</book> IN $book, y > 1991
CONSTRUCT <result>
<title>$t</>
WHERE $a2 IN $a
CONSTRUCT <author>$a</>
</result>
</results>
```

XML-QL patterns are positional, are capable of retrieving multiple data items at once from multiple sources and evaluate joins over variable name equality. XML-QL does not differentiate between ordered and unordered queries (everything is matched in any order). XML-QL can construct new data, but grouping can only be achieved by using nested sub-queries (as in the example above). XML-QL supports aggregations. Queries are rule-based, and sub-queries can be nested into the query rules (as above). Construction and querying are separated, but this separation is abandoned when using nested sub-queries. XML-QL is not aware of type information.

4.2.2.2 UnQL

UnQL [93], the *Unstructured Query Language* is a query language originally developed for querying semistructured data. UnQL uses a positional, pattern-based selection and a query consists of a single `select ...where ...` rule which separates construction from querying. Queries may also be nested, in which case the separation of querying and construction is abandoned. UnQL uses its own, non-XML syntax for representing and querying graph-structured data.

Query 17 (XML-QL). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
select { results: (
select { result: { title: T,
( select { author: A }
where { author: A } in Book )
}
}
where { book: Book } in Bib,
{ year: Y, title: T } in Book ) },
Y > 1991
where { bib: Bib } in db
```

In terms of properties, UnQL is very similar to XML-QL: it uses positional patterns, can retrieve multiple data items at once and from multiple sources, joins are possible over variable name equality. However, UnQL can respect the order of data, if desired. Like in XML-QL, construction is possible, but grouping can only be achieved by using nested sub-queries. Aggregations are supported. Queries are rule-based, and sub-queries can be nested into the query rules (as in the example above). Construction and querying are separated, but this separation is abandoned when using nested sub-queries. Like XML-QL, UnQL is not aware of type information.

4.2.2.3 XML-RL

XML-RL [249] is a proposal for a pattern-based query language based on logic programming. Patterns are expressed by terms that may contain logic variables and may be partly abbreviated with a path syntax similar to XPath. An XML-RL query program consists of several rules denoted by $A \Leftarrow L_1, \ldots, L_n$ where A is used for construction and $L_1, \ldots, L_n$ are query patterns. Rules may interact via rule chaining and it is possible to use recursion.

Query 18 (XML-RL). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
(file:result.xml)
/results/result: (title: $t, {author: $a})
⇐
(file:bib.xml)
/bib/book: (@year: $y, title: $t, author: $a), $y > 1991
```

XML-RL is a positional language, but allows path expressions as abbreviations. It can retrieve multiple data items from multiple sources and supports joins via variable name equality, as XML-QL and UnQL. It is not possible to query based on order. XML-RL can construct new data items. Grouping is possible by means of a special construct `{.}` and aggregations are supported. Queries programs can be structured by using more than one rule, and rules may interact via rule chaining. Querying and construction are always separated. However, computations are usually performed in the query part. XML-RL does not support type information.

4.2.2.4 XMAS

XMAS [251] part of MIX [30], the *XML Matching And Structuring language*, is a declarative, rule-based query language for XML. Rules are of the form `CONSTRUCT ...WHERE ...` and resemble XML-QL rules very closely. However, XMAS provides more powerful constructs for grouping/collection and aggregation, in a similar way to the grouping construct `{.}` of XML-RL.

Query 19 (XMAS). Select all authors and titles of books written after 1991 and return them in `result` elements contained within a `results` element.

```
CONSTRUCT
<results>
<result>
$T
$A {$A}
</result> {$T}
<results>
WHERE
<bib>
<book year=$Y>
$T: <title/>
$A: <author/>
</>
</> IN "bib.xml"
AND $Y > 1991
```

Note the grouping expressed by enclosing the collected variables in curly braces. For every instance of `$T`, a result element is created. Within every such result element, all authors are collected.

Like XML-QL, XMAS employs positional, rule-based selection, can query multiple data items at once and from multiple sources. In contrast to XML-QL, joins have to be expressed using an explicit join operator. XMAS is not capable of querying the order of elements, and does not support incompleteness in depth. New data can be constructed in rule heads using sophisticated grouping constructs that avoid nested subqueries. Apparently, aggregations and subqueries are not supported. Construction and querying is therefore always separated. XMAS is not aware of type information.

4.2.2.5 XET/XDD

XET [17], *XML Equivalent Transformations*, is a pattern-based, rule-based query language for XML aiming primarily at Semantic Web applications, but also capable of querying generic XML data. XET employs XML-based patterns enriched with variables for retrieving data items in XML documents. XET rules are similar to rules in logic programming and support rule chaining. A formal semantics is provided in form of *XDD* (*XML Declarative Description*) and *ET* (*Equivalent Transformations*). Unfortunately, there is not enough information available to express the running example in XET.

4.2.2.6 Xcerpt

Xcerpt [329] is a pattern-based query language for XML inspired by logic programming. It is further extended in the REWERSE I4 working group following the design principles detailed in [83]. An Xcerpt program consists of at least one *goal* and some (possibly zero) *rules*. Rules and goals contain query and construction patterns, called *terms*. Terms represent tree-like (or graph-like) structures. The children of a node may either be *ordered*, i.e. the order of occurrence is relevant (e.g. in an XML document representing a book), or *unordered*, i.e. the order of occurrence is irrelevant and may be chosen by the storage system (as is common in database systems). In the term syntax, an *ordered term specification* is denoted by square brackets `[ ]`, an *unordered term specification* by curly braces `{ }`.

Likewise, terms may use *partial term specifications* for representing incomplete query patterns and *total term specifications* for representing complete query patterns (or data items). A term *t* using a partial term specification for its subterms matches with all such terms that (1) contain matching subterms for all subterms of *t* and that (2) might contain further subterms without corresponding subterms in *t*. Partial term specification is denoted by *double* square brackets `[[ ]]` or curly braces `{{ }}`. In contrast, a term *t* using a total term specification does not match with terms that contain additional subterms without corresponding subterms in *t*. Total term specification is expressed using *single* square brackets `[ ]` or curly braces `{ }`.

Data Terms represent XML documents and the data items of a semistructured database, and may thus only contain total term specifications (i.e. single square brackets or curly braces). They are similar to *ground* functional programming expressions and logical atoms. A *database* is a (multi-)set of data terms (e.g. the Web). A non-XML syntax has been chosen for Xcerpt to improve readability, but there is a one-to-one correspondence between an XML document and a data term.

Query Terms are (possibly incomplete) patterns matched against Web resources represented by data terms. They are similar to the latter, but may contain *partial* as well as *total* term specifications, are augmented by *variables* for selecting data items, possibly with *variable restrictions* using the $\leadsto$ construct (read *as*), which restricts the admissible bindings to those subterms that are matched by the restriction pattern, and may contain additional query constructs like *position matching* (keyword `position`), *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`).

Query terms are "matched" with data or construct terms by a non-standard unification method called *simulation unification* that is based on a relation called *simulation*. In contrast to Robinson's unification (as e.g. used in Prolog), simulation unification is capable of determining substitutions also for incomplete and unordered query terms. Since incompleteness usually allows many different alternative bindings for the variables, the result of simulation unification is not only a single substitution, but a (finite) *set of substitutions*, each of which yielding ground instances of the unified terms such that the one ground term matches with the other.

Construct Terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by *variables* (acting as place holders for data selected in a query) and the *grouping construct* `all` (which serves to collect all instances that result from different variable bindings). Occurrences of `all` may be accompanied by an optional sorting specification.

Example. *Left:* A query term retrieving departure and arrival stations for a train in the train document. Partial term specifications (partial curly braces) are used since the train document might contain additional information irrelevant to the query. *Right:* A construct term creating a summarised representation of trains grouped inside a `trains` term. Note the use of the `all` construct to collect all instances of the `train` subterm that can be created from substitutions in the substitution set resulting from the query on the left.

```
travel {{
train {{
departure {{
station { var From } }},
arrival {{
station { var To } }}
}}
}}
```

```
trains {
all train {
from { var From },
to { var To }
}
}
```

Xcerpt also provides means for defining (possibly recursive) rules. Based upon these ability, an library of view definitions for convenient querying of RDF (and Topic Maps) is currently in development.

4.3 RDF Query Languages

4.3.1 SquishQL-family

4.3.1.1 SquishQL

SquishQL [282, 281] is an RDF query language that has been developed with ease-of-use and similarity to SQL as main principles. It has been implemented in at least three different processors, most notably in a slightly modified and further refined version, named RDQL, in the Jena Toolkit [190].

Essentially, SquishQL aims at a more intuitive, easy-to-use and fast-to-learn means for accessing RDF triples than provided by general-purpose RDF APIs. This aim also leads to a certain restriction with respect to the supported features.

The SquishQL query model is influenced by [196] and uses so-called "triple patterns" and conjunctions of these "triple patterns" to specify the structure of the RDF graph to be matched by the query. As stated in [282] "this results in quite a weak pattern language but it does ensure that in a result all variables are bound."

To give an impression of the syntax used by SquishQL, the following example shows Query 1 formulated in SquishQL:

```
SELECT ?essay, ?author, ?authorName
FROM http://example.org/books
WHERE (?essay, <rdf:type>, <books:Essay>),
(?essay, <books:author>, ?author),
(?author, <books:name>, ?authorName)
USING books FOR http://example.org/books#,
rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Since RDQL is based on and mostly identical to SquishQL, refer to Section 4.3.1.3 for more information on this language family.

Project page:

– (Inkling: `http://swordfish.rdfweb.org/rdfquery/`, RDFStore: `http://rdfstore.sourceforge.net/`)

Implementation(s):

Inkling [281], Jena Toolkit—RDQL [334, 332, 333], RDFStore [319]

Online demonstration:

`http://demo.asemantics.com/rdfstore/www2003/`

4.3.1.2 rdfDB Query Language

rdfDB [195] is an early proposals for an RDF data base that influenced the design of, among others, SquishQL and RDQL (cf. Section 4.3.1.3). [152] gives an introduction into rdfDB, more details on the query language can be found in [195].

The syntax of rdfDB is SQL-like and as in SQL several different database commands can be executed in a single session, although no transactions management is provided. Aside of commands for creating databases, inserting and deleting triples, and defining namespaces, the core element of the rdfDB syntax is the `select-from-where` clause. As all database commands such a clause is delimited by a `</>` and returns bindings for any number of variables specified after the `select` such that the pattern specified in the `where` clause matches in the database given in the `from` part.

For illustration of rdfDB consider again Query 1 and a possible implementation of that query in the rdfDB query language (assuming the content of `http://example.org/books` has been stored in a database called `booksdata`):

```
enter namespace xmlns:books http://example.org/books# </>
enter namespace xmlns:rdf http://www.w3.org/1999/02/22-rdf-syntax-ns# </>
select ?essay, ?author, ?authorName from booksdata
where (rdf:type ?essay books:Essay), (books:author ?essay ?author),
(books:name ?author ?authorName) </>
```

rdfDB is considered influential on the design of SquishQL and RDQL discussed in this section, but the development seems to have been halted. Nevertheless, an early implementation of rdfDB is still available from the project page.

Project page:

`http://www.guha.com/rdfdb/`

Implementation(s):

prototype available from the project page

Online demonstration:

none

4.3.1.3 RDQL

RDQL, RDF Data Query Language, is a query language for RDF models developed by Andy Seaborne at HP and recently submitted to the W3C as candidate for standardization [332,282,334,333]. It is an evolution from several languages, inspired by the work in [196]: rdfDB [195], SquishQL [282], and Inkling [281].

As SquishQL RDQL does not support the special meaning provided by the RDF schema level, although at least one of its implementations (viz. in the Jena Toolkit [190]) provide a transparent transitive closure over the subsumption hierarchies defined in RDFS using *rdfs:subClassOf* or `rdfs:subPropertyOf`.

A typical RDQL query has a syntax which is reminiscent of SQL, but which is built around a set of conjuncted triple patterns, i.e. triples of constants or variables, which are resolved on the graph. Such a process binds the variables to node or property labels, which are URIs and constants, and the result of the query is a subset of those bindings. Additional constraints on variable values can be used to filter the result. For instance, the query:

```
SELECT ?person
FROM http://somewhere.org/some-rdf-model-of-people
WHERE (?person, <http://example.org/peopleInfo#age>, ?age)
AND ?age > 24
```

returns the URIs of persons whose age is greater than 24.

`SELECT` is the only language statement, and its syntax is easily described:

`SELECT`	variables (identifies the variables whose bindings are returned)
`FROM`	model URI
`WHERE`	list of triple patterns
`AND`	boolean expression (the filter to be applied to the result)
`USING`	name `FOR` uri, ...

The last clause allows the simplification of queries by introducing names for long URIs. For instance, the previous query can be rewritten as:

```
SELECT ?person
FROM http://somewhere.org/some-rdf-model-of-people
WHERE (?person, info:age, ?age)
AND ?age > 24
USING info FOR <http://example.org/peopleInfo#>
```

The language is maintained intentionally simple, operating only on the "data" level of RDF, so that it could be easily amenable to standardization as a "low-level" RDF language, which relies on higher level services to make use of rules or inference facilities. As the author explicitly states, "if a graph implementation provides inferencing to appear as 'virtual triples' (i.e. triples that appear in the graph but are not in the ground facts), then an RDQL will include those triples as possible matches in triple patterns", so that no distinction is made between inferred triples and ground triples [332].

This language philosophy, moreover, has the effect that the predefined properties which describe semantic oriented or "schema" aspects of a model, like type, set or class relations, are treated as ordinary properties, leading to cumbersome complex queries when those aspects are involved (like, for instance, a query to return all the elements of a container). This problem, too, could be solved by using other specialized wrappers around a model.

In the following, we will consider the main aspects of the language according to the classification criteria previously shown.

Easy of use The queries are fairly simple to write and understand, although the language has no visual syntax. Its support of the "natural" graph RDF model, with the simple triple patterns, and a SQL-like syntax, make the language easy to grasp, even for persons non experts of all the intricacies of the semantic web languages and models. Due to the language simplicity, no modularization or extension mechanisms exist for writing complex queries, but the language can be used inside the Java programming language, even mixing it, at a certain extent, with low-level calls to the model's API.

Functionality—Query Types The language supports only selection and extraction, since the result of a query is a set of bindings based on triple patterns matching and filtering: no kind of "data restructuring" is possible, nor the building of new data. Only the basic data level of RDF is supported, and the programmer must cope with the specialized constructs of the framework like containers, reification, optional properties, as they were ordinary properties. For instance, the following query extracts all the elements from the bag (a kind of container) identified by `http://somewhere.org/bag1`.

```
SELECT ?y
WHERE (<http://somewhere.org/bag1>, ?x, ?y)
AND ! ( ?x eq rdf:type && ?y eq rdf:Bag)
```

The filter part is necessary to eliminate from the result the triple which simple states that the resource is a bag (i.e. has the property `rdf:type` equal to `rdf:Bag`). Queries cannot be compound, and have a single input and output. The filtering part allows the use of simple types URIs, strings, numbers, booleans and the null value, with the corresponding operators. The negation can be used (like in the previous example). On the other hand, the list of triples are only positively conjuncted: no disjunction, negation or optional matching is allowed. This, while severely limiting the expressive power of the languages, has the consequence that a query result is always a set of bindings of values to variables (and not, for instance, subgraphs). Another important limitation is that, although a variable can be bound to a blank node, there is no way to specify in a triple that a node is a blank one, neither with a literal nor with a variable. So, for instance, it is not possible to ask a query which returns all the blank nodes of a graph. No form of recursion or iteration is allowed: only paths of definite lengths can be queried by listing explicitly all the triples forming the path. Finally, no modification to the data can be carried through the language.

Considering the sample queries from Section 3.2, only the first two queries can be expressed in RDQL. Query 1 could be formulated (exactly as in SquishQL above) as

```
SELECT ?essay, ?author, ?authorName
FROM http://example.org/books
WHERE (?essay, <books:author>, ?author),
(?author, <books:authorName>, ?authorName)
USING books FOR http://example.org/books#,
rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Note that everything with an author is considered to be an essay. Otherwise we should add a triple pattern like the following: `(?essay, rdf:type, books:Essay)` if each book is classified as an element of `books:Essay`. Since RDQL is not ontology-aware and no recursion or other mechanism for computing the transitive closure of the subsumption hierarchy is available, it is not possible to select all resources classified as an element of some class that is a sub-class of `books:Essay`.

For Query 2 a similar problem arises. A first version in RDQL could be

```
SELECT ?property, ?propertyValue
FROM http://example.org/books
WHERE (?essay, <books:book-title>, "Bellum Civile")
(?essay, ?property, ?propertyValue),
USING books FOR http://example.org/books#
```

Note that a property value could be a node with other properties: however, since no recursive mechanism is available in the language, we cannot express a transitive closure of all such properties.

As part of the RDQLPlus (`http://rdqlplus.sourceforge.net/`) implementation of RDQL, an language extension called RIDIQL [373] is defined providing both updates and transparent use of the inference abilities of the underlying Jena Toolkit [190].

Semantics No formal semantics has been published for RDQL.

Complexity and implementation The RDQL has several implementations, of which a well known one is that found in the comprehensive Jena package for semantic web developed in Java at HP Labs [190]. It can work both with an in-memory representation of an RDF model, as well as, for high scalability, with a database based one (currently for MySql, Oracle E, Postgres). The database representation allows for efficient retrieval of triples by storing them in denormalized tables. The table fields are indexed so that the pattern matching engine can retrieve the triples by using the constants as keys for the search. On the other hand, the filtering part of the query is evaluated in memory on the resulting tuples. Such an approach, while not maximizing the performances of the system, allows the query engine to be implemented with limited complexity.

No formal complexity study of the language has been published so far.

Reasoning No reasoning mechanism is present in the language.

Ontology awareness As already specified, the language ignores every kind of ontological aspect, including typing mechanisms. If such aspects must be considered, they must be treated like all the user-defined data.

Project page:

`http://www.hpl.hp.com/semweb/rdql.htm`

Implementation(s):

Jena Toolkit [190], RAP (RDF API for PHP) [292], PHP XML Classes (`http://phpxmlclasses.sourceforge.net/`), RDFStore [319], Rasqal RDF Query Library (`http://www.redland.opensource.ac.uk/rasqal/`), Sesame (`http://www.openrdf.org/index.jsp`), 3store (`http://sourceforge.net/projects/threestore/`, cf. [202]), RDQLPlus (`http://rdqlplus.sourceforge.net/`)

Online demonstration:

using Sesame: `http://www.openrdf.org/demo.jsp`
using RAP: `http://www3.wiwiss.fu-berlin.de/rdfapi-php/test/custom_rdql_test.php`
using RDFStore: `http://demo.asemantics.com/rdfstore/www2003/`

4.3.1.4 BRQL

BRQL [315] has been recently developed by members of the W3C "RDF Data Access" Working Group as an extension of RDQL [332] aligned with the requirements and use cases detailed in [124]. It is still a very early draft and constantly being improved, therefore this evaluation can only give an impression of the current status.

Several features missing from RDQL but identified as interesting or necessary for an RDF query language in [124] are added, most notably:

- The ability to construct a (single) new RDF graph using the `CONSTRUCT` keyword. The new graph can be specified with RDQL triple or graph "patterns".

- A query using the DESCRIBE clause returns the "description" of resources matched by the query part of the expression. The exact meaning of "description" is not yet defined.
- In contrast to RDQL, BRQL supports convenient querying for "quads", i.e., triples with context information such as source attribution.
- BRQL provides the keyword OPTIONAL to specify triple or graph "patterns" that should be attempted to match but where failure to match does not cause a query solution to be rejected.
- Finally, also a means for testing the non-existence of tuples is added to the language.

For illustrating the capabilities of BRQL, consider again Query 1. But additionally we also would like to return any translator of book, if there is any. This can be expressed in BRQL as

```
SELECT ?essay, ?author, ?authorName, ?translator
FROM http://example.org/books
WHERE (?essay books:author ?author),
(?author books:authorName ?authorName)
OPTIONAL (?essay books:translator ?translator)
USING books FOR http://example.org/books#,
rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Thanks to the addition of the CONSTRUCT clause, also restructuring and non-recursive inference queries can be expressed. Query 4 can be implemented by the following expression

```
CONSTRUCT (?y books:authored ?x)
FROM http://example.org/books
WHERE (?x books:author ?y)
USING books FOR http://example.org/books#
```

and Query 9 by

```
CONSTRUCT (?x books:co-author ?y)
FROM http://example.org/books
WHERE (?book books:author ?x)
(?book books:author ?y)
AND (?x neq ?y)
USING books FOR http://example.org/books#,
```

Project page:
http://www.w3.org/2004/07/08-BRQL/

Implementation(s):
none

Online demonstration:
none

4.3.1.5 TriQL

TriQL [56] is under development at the Freie Universität Berlin, German, and aims to extend RDQL to query named graph as introduced in TriG [55] by the authors of TriQL. The reasoning for introducing named graphs in the RDF data model is given in [101].

The use of named graphs allows, e.g., the grouping of assertions by source or author. Then a query such as "Get all books with rating above a threshold of 5. Use only information, that has been asserted by Marcus Tullius Cicero." can be formulated as

```
SELECT ?books
WHERE ?graph ( ?books books:rating ?rating )
(?graph swp:assertedBy ?warrant)
(?warrant swp:authority <http://people.net/cicero>)
USING books FOR http://example.org/books#,
swp FOR <http://www.w3.org/2004/03/trix/swp-1/>
```

Project page:

`http://www.wiwiss.fu-berlin.de/suhl/bizer/TriQL/`

Implementation(s):

none

Online demonstration:

none

4.3.2 Query Languages influenced by XPath, XSLT or XQuery

In this section, a number of query languages are discussed that have been influenced or are extensions of existing XML query languages developed by the W3C. Some of these approaches (viz. [324, 350, 366]) can be implemented directly on top of theses languages merely defining some extension functions and data normalization to be applied before querying. The others propose query languages for RDF that are in spirit and syntax similar to the XML query languages mentioned above.

4.3.2.1 XQuery for RDF: “The Syntactic Web” Approach

During the initial development on XQuery 1.0 [57], Jonathan Robie et al. proposed in a series of articles [321,324] the use of XQuery for processing RDF. The issue of normalizing RDF before querying is discussed in detail (cf. Section A.3.4). Based on a suitable normal form (essentially statements are considered as triples but statements with same subject are grouped), it is shown how XQuery can be used to query such normalized RDF.

Support for the special semantics of the properties defined in RDFS is added by means of several functions. E.g., the function `rdf:instance-of-class` computes the sequence of all resources (represented by their `description` element) that are an instance of a given class or any of its sub-classes. This is achieved by the following recursive function definition (the first parameter is the set of resources on which to operate):

```
define function
rdf:instance-of-class($t as element(description)*,
$base-name as xs:string)
as element(description)*
{
$t[rdf:type = $base-name]
,
for $i in $t[rdfs:subClassOf = $base-name]
return rdf:instance-of-class($t, string($i/@rdf:about))
}
```

Using this function, Query 1 could be formulated (assuming an appropriate normalization has been applied to the RDF data) as:

```
let $t := document("http://example.org/books")//description
for $essay in rdf:instance-of-class($t, "books:Essay"),
$author in $t[rdf:about = $essay/books:author]
```

```
return
<result>
{$essay, $author}
</result>
```

The result of such a query is a sequence of `result` elements containing an essay and one of its author. The name of the author does not need to be queried specifically, since that information is already provided as part of the description of the author selected in the `$author` variable. The query also illustrates that the approach of implementing RDF querying on top of an XML query language has the virtue of being able to return the result of an RDF query in an arbitrary XML format.

The approach also covers the normalization and querying of Topic Maps. Similar to RDF specialized functions are defined to support the specificities of the Topic Maps data model, e.g., the following function computes all derived classes defined in a Topic Maps:

```
define function tm:get-derived-classes($topics as element(topics)*,
$derivations as element(associations)*, $base as element(topic))
as element(topics)*
{
let $a := tm:get-association-by-topic-role($topics,
$derivations, $base, "superclass")
for $subclass in tm:get-topic-playing-role($topics,
$a, "subclass", ())
return (
$subclass,
tm:get-derived-classes($topics, $derivations, $subclass)
)
}
```

Project page:
none

Implementation(s):
can use any XQuery implementation, however function library has not been made available, some functions are given in [324].

Online demonstration:
none

In [305] it is strongly argued that a unified model of RDF and XML data and a query language based upon such a model is essential for the success of the Semantic Web vision. It is demonstrated, that although the RDF and XML data models differ in some points, a common model theoretic interpretation of RDF and XML data is possible. However, no query language has been proposed based upon this work.

Recently, some information about another approach for extending XQuery for RDF querying, called REX "RDF Extensions to XQuery", has been discussed in the W3C Data Access Working Group (cf. [338]). This approach seems to be similar to the one discussed above, however RDF statements are generated on-demand by a specific function `related(`*subject*, *predicate*, *object*`)`. Also no support for RDFS seems to be provided yet. Due to lack of information this extension is not considered further in this survey.

4.3.2.2 XsRQL: An XQuery-style RDF Query Language

XsRQL (XQuery-style RDF Query Language) [222] is a very recent proposal for a RDF query language that borrows from XQuery 1.0 [57], both with respect to the syntax and the design approach. The main objectives of the language design are simplicity and flexibility. In particular, the language aims at providing a syntax flexible enough to allow both the writing of rather simple, concise and more complex but also more expressive queries.

At the core of the proposal are two main differences from XQuery:

- The data model should be adapted to the specificities of RDF. The current draft is rather vague on this point. Some issues can be inferred from the examples given.

- The path language used for accessing and selecting nodes in the data structure has been adapted to the RDF data model: Essentially the same syntax as for XPath is used, however only the `child` axis is supported. Properties are separated from subjects and objects by using the attribute indicator `@` from XPath. However, in contrast to XML attributes, the values (i.e., objects of statements) of RDF properties are not simple, but rather structured values. Therefore after an property further steps may follow in a path expression.

Consider once again Query 1. In XsRQL that query could be stated as

```
declare prefix books: = <http://example.org/books#>;
declare prefix rdf: = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>;
for $essay in datasource( <http://example.org/books>)//*[@rdf:type/books:Essay],
$author in $essay/@books:author/*
return
$essay, $author, $author/@books:authorName/*
```

As XsRQL currently neither supports a closure operation, a descendant-like operator or some other means of traversing an arbitrary-length path in the data structure, it is not possible to return also resources classified by any sub-class of *books:Essay*.

Project page:
`http://www.fatdog.com/xsrql.html`

Implementation(s):
none

Online demonstration:
none

4.3.2.3 XSLT for RDF: TreeHugger and RDF Twig

Similar in spirit to the approaches discussed in Section 4.3.2.1, **TreeHugger** [350] allows the querying and transformation of RDF data in XSLT. However, in contrast to [324] (and due to limitations of XSTL 1.0), the normalization is performed by means of XSLT extension functions and not by an XSLT program. Also the normal form of RDF used for querying is based on the RDF striped syntax [73], but properties are represented both as XML elements and as attributes (raising some problems for multi-valued properties). Three extension functions are provided, one for loading an mere RDF document, one for loading an RDF document and handling the special vocabulary defined by RDFS, and one for loading an RDF document and handling the vocabulary of both RDFS and OWL.

For accessing nodes in an RDF document XPath is used with a special prefix `inv` that allows querying the inverse of a property.

Query 1 could be expressed by the following XSLT stylesheet with TreeHugger extensions:

```
<results xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:books="http://example.org/books#"
xmlns:th="http://rootdev.net/net.rootdev.treehugger.TreeHugger"
xmlns:rdfs="http://www.w3.org/2000/01/rdf−schema#"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xsl:version="1.0">
<!-- Load RDF document -→
<xsl:variable name="doc"
select="th:documentRDFS('http://example.org/books')" />
<xsl:for-each select="$doc/books:Essay">
<xsl:for-each select="books:author/*">
<result>
<xsl:value-of select="inv:books:author" />
<xsl:value-of select="." />
<authorName>
<xsl:value-of select="books:authorName/*" />
</authorName>
</result>
</xsl:for-each>
</xsl:for-each>
</results>
```

Project page:

`http://rdfweb.org/people/damian/treehugger/`

Implementation(s):

available from the project page

Online demonstration:

`http://swordfish.rdfweb.org/discovery/2003/09/treehugger/`

In [366], another approach of extending XSLT 1.0 with functions for querying RDF, called **RDF Twig**, is described. In contrast to the previously discussed proposals, it provides different views on the RDF data corresponding to redundant or non-redundant (i.e., where nodes that are reachable by various paths are repeated, resp. not repeated) depth or breadth first traversals of the RDF graph. Furthermore, two query mechanisms are provided: A small set of logical operations on the RDF graph (also used in the example below) and an interface to the RDQL query engine provided by the Jena Toolkit [190] used for implementing RDF Twig.

To give a feeling for the language, we consider once more Query 1 and how it can be realized in this query language:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
xmlns:rt="http://nwalsh.com/xslt/ext/com.nwalsh.xslt.saxon.RDFTwig"
xmlns:twig="http://nwalsh.com/xmlns/rdftwig#"
xmlns:books="http://example.org/books#"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#">
<xsl:template match="/">
<xsl:variable name="model"
select="rt:load('http://example.org/books')"/>
<!-- this is used as default model from now on-→
<xsl:variable name="pType"
select="rt:property('http://www.w3.org/1999/02/22−rdf−syntax−ns#','type')"/>
<xsl:variable name="essays"
select="rt:find($label, 'books:Essay')"/>
<xsl:variable name="tree"
select="rt:twig($essays)/twig:result"/>
<results>
<xsl:for-each select="rt:find($label, 'books:Essay')">
<result>
<xsl:value-of select="rt:twig(.)" />
<xsl:value-of select="rt:twig(.)/twig:result/books:author" />
</result>
```

```
</xsl:for-each>
</results>
</xsl:template>
```

For simplicity, only essays and authors are considered, the names of the authors will be returned as well, as they are reachable from the associated essay and `rt:twig(.)` returns all information reachable from the current essay (in this case). RDF Twig does not support RDFS or OWL, therefore only resources classified directly as `books:Essay` will be considered by this query.

Project page:
`http://rdftwig.sourceforge.net/`

Implementation(s):
available from the project page

Online demonstration:
none

4.3.2.4 RDFT and Nexus Query Language: XSLT-style RDF Query Languages

RDFT As XsRQL is modeled after XQuery, RDFT [136] is a draft proposal closely related to XSLT 1.0. As XSLT 1.0 it uses templates that are matched recursively against the data structure. Naturally, the structural recursion is performed against an RDF graph, raising issues with cyclic graph structure that are still open issues in the development of RDFT.

RDFT uses an adaptation of XPath for querying RDF graphs, called NodePath. As in most of the other approaches oriented on XML query languages, a striped view of RDF [73] is adapted where properties and other resources alternate. No provision for querying data described with RDFS or OWL is made.

Only a subset of XSLT elements is supported, but a macro mechanism is introduced as illustrated in the following implementation of Query 1 in RDFT:

```
<rt:stylesheet rt:version="1.0" xmlns:rt="http://purl.org/vocab/2003/rdft/">
<rt:macro-set rt:prefix="rdf">
<rt:macro name="type"
value="resource('http://www.w3.org/1999/02/22−rdf−syntax−ns#type')/resource()"/>
</rt:macro-set>
<rt:root-template>
<rt:apply-templates
rt:select="/resource()[rdf:type = resource("http://example.org/books#Essay")/>
</rt:root-template>
<!-- Template for the Essay
<rt:template pattern="resource()[rdf:type =
resource('http://example.org/books#Essay')" />
<xsl:value-of select="." />
<rt:apply-templates
rt:select="resource('http://example.org/books#author')/resource()/>
</rt:template>
<!−− Template for the author −→
<rt:template
pattern="resource('http://example.org/books#author')/resource()">
<xsl:value-of select="." />
</rt:template>
</rdft:stylesheet>
```

Again, for simplicity, only books and their authors are returned without considering the names of the authors. Also note, that the specification is not really clear what the result of such a query will be: an XML tree or some form of an RDF graph. The description of `rt:element` seems to indicate the former, the description of `rt:value-of` the latter.

Project page:
`http://www.semanticplanet.com/2003/08/rdft/spec`

Implementation(s):
none

Online demonstration:
none

Nexus Query Language In [3] another approach for querying RDF (and some form of XML) using an XSLT-like language is sketched. The basic idea is to translate both RDF/XML and also some non-RDF XML documents into a hierarchy of elements (that also carry some attributes) based upon the relations between the elements. The result of a query is then some (also hierarchical) view over this element tree. [3] gives no consideration w.r.t. cyclic relations among elements but the language used seems to indicate that only proper hierarchies can be represented.

RDF statements are mapped to nodes in the data model in the following way: nodes in the graph represent RDF properties; an RDF statement (S, P, O) is represented by edges from all nodes representing some property with the value S to a node representing the property P with value O. A resource that never occurs as object is assigned as value to a special property called `query:seed`. [3] seems to indicate that there can be only one such `query:seed` node, an assumption that is clearly invalid for general RDF graphs.

The query language provides means for matching such property nodes based on the identifier (represented as URI or XML QName) of the property and the type (as determined by an `rdf:type` statement) of the value of the property.

Consider again Query 1 and the following Nexus query that implements the query:

```
<query:plan>
<query:template match="query:seed" type="books:Essay">
<query:call name="query:insert" rename="book">
<query:call name="query:format" rename="title"
value="book:title" />
<query:call name="query:traverse" />
</query:call>
</query:template>
<query:template match="book:author">
<query:call name="query:insert" rename="author">
<query:call name="query:format" rename="name"
value="book:authorName" />
</query:call>
</query:template>
</query:plan>
```

An excerpt of the result of this query on the sample data from Figure 2.1 would be:

```
...
<book title="Bellum Civile">
<author name="Julius Caesar" />
<author name="Aulus Hirtius" />
</book>
...
```

Obviously, the syntax is rather verbose and does not inherit the ability of XSLT to write arbitrary XML as content of an element. Furthermore, the development of the language seems to be stalled as the only information available on this language is the rather short and often vague report cited above [3].

Project page:
none

Implementation(s):
: not publicly available, no report on any implementation

Online demonstration:
: none

4.3.2.5 XPath-syle Access to RDF: RDF Path, RPath and RxPath

Several mostly sketchy proposals [352, 137, 302, 267]for adapting the XPath-style navigational access to RDF graphs have been published in recent years. For this survey, two representative ones have been selected, viz. "Pondering RDF Path" by Sean Palmer and RPath [267], a language designed in the context of device independence and content adaptation.

RDF Path In [302], a sketch of an RDF Path language is proposed that is closely aligned with XPath. The aim of this language is to provide clear equivalents for the XPath facilities such as selection of context nodes, filtering and location steps, but operating on RDF data. The syntax is similar to XPath but extended by special node-tests for RDF, such as `arc()` and `subj()` for selecting all arcs, resp. all subjects in the RDF data. Several aspects of XPath not relevant for RDF are dropped. Only basic navigational features are provided, functions and testing of values is mostly not considered in this early draft. Finally, as in most of the approaches based on XML query languages, the fact that, in contrast to XML trees, RDF graphs are not rooted is not considered.

As XPath, this language is not capable to select related information as in Query 1. Therefore a slightly variation of Query 2 is used to illustrates its abilities: "Select the names of all authors of historical essays with the title 'Bellum Civile'". This query can be realized by the path expression

```
*[rdf:type/books:Historical_Essay books:title/"Bellum Civile"]/
books:author/*/books:authorName
```

Appropriate mappings for the prefixes used in the query have to be established before evaluating the query (this is also the case in XPath). Also note that due to the lack of support for the special semantics of the vocabulary provided by RDFS, this query will only return resources directly classified as `books:Historical_Essay`.

Project page:
: `http://infomesh.net/2003/rdfpath/`

Implementation(s):
: none

Online demonstration:
: none

RPath [267] is a new RDF query language which is based on the *path navigation* principle known from XPath. In fact, the major design idea behind RPath is to provide *XPath for RDF*. In its current state, however, it focuses strongly on CC/PP and UAProf, two RDF applications for describing device characteristics like *color capable*, *color depth*, or *screen resolution*. CC/PP is the general framework for those *device profiles*, whereas UAProf is a specific vocabulary focused on (but not limited to) mobile devices.

As RPath navigates through RDF data using paths, it views this data as graph, not as triples. The concepts of the language resemble very much that of XPath, that is, location steps, *vertex-edge-tests* (corresponding to node-tests in XPath), and predicates. Differences to XPath are due to the differences between the data models of XML and RDF, for example: The axes can follow a path along vertices (RDF predicates) and edges (RDF subjects and objects). The adaptation of most XPath concepts to the RDF data model is straightforward. One major difference, however, is the absence of a root node in the RDF graph, and the question of finding a start point for the path expression remains an open topic.

On the other hand, the (current) focus of RPath lies on CC/PP and UAProf, which limit RDF graphs to rooted two-level trees. The language itself does not offer any CC/PP-specific or UAProf-specific features, all of it can be used to query generic RDF graphs. The implementation of the prototype, however, is capable of handling the protocol specified by CC/PP, including *default profiles* and *profile diffs*, as well as the data types defined by UAProf.

The authors of RPath claim its ease of use as the main advantage. As XPath already enjoys widespread use in the XML world, the learning curve for RPath should be shallower than that for other RDF query languages for potential users. Another advantage is the tight coupling with CC/PP and UAProf, which should make RPath suitable to be used in *device independence*-applications.

RPath has been developed at Keio University, Japan, by Keita Matsuyama, Michael Kraus, Kazuhiro Kitagawa (Activity lead for the W3C Device Independence), and Nobuo Saito. Currently, development has been halted.

To convey an impression of the languages capabilities and syntax, the same variation of Query 2 as above is used: “Select the names of all authors of historical essays with the title ‘Bellum Civile’.”.

```
/@vertex()[
rdf:type/@books:Historical_Essay and
books:title/@vertex()[equals(‘Bellum Civile’)]
]/books:author/books:authorName
```

Note, that in contrast to most other path-based approaches to querying RDF data, RPath does not require the user to write paths where expressions matching vertices (i.e., classes) and edges (i.e., properties) alternate (similar as in striped RDF [73]). This is possible, as all steps begin with an axis specification and different axes for vertices and edges are provided. To illustrate this point the same query shown above could be written using non-abbreviated RPath syntax as:

```
outerVertex::vertex()[
outEdge::rdf:type/outVertex::books:Historical_Essay and
outEdge::books:title/outVertex::vertex()[equals(‘Bellum Civile’)]
]/outEdge::books:author/outEdge::books:authorName
```

Project page:
none

Implementation(s):
prototype in Java, based on a CC/PP engine from Sun

Online demonstration:
none

RxPath As part of the Rx4RDF project `http://rx4rdf.liminalzone.org/rx4rdf` that aims at improving the accessibility of RDF for non-experts another adaption of XPath for querying RDF data has been defined.

In contrast to the approaches discussed above and somewhat related to TreeHugger and RDF Twig, RxPath is essentially " a mapping between the RDF Abstract Syntax to the XPath Data Model" [346]. The mapping consists in four steps:

- One top-level element in the XML document is created for every resource in the RDF model with the type of the resource as element label.

- "Each root element has a child element for each statement the resource is the subject of. The name of each child is [the] name of the property in the statement." [345]

- "Each of these children have [a] child text node if the object of the statement is a literal or a child element if the object is a resource." [345]

- "Object elements have the same name and children as the equivalent root element for the resource, thus defining a potentially infinitely recursive tree." [345]

As stated, such a mapping might lead to infinite trees, in particular when evaluating any of the closure axes of XPath (`descendant`, `following`, `preceding`, etc.) the number of nodes selected in the tree is no longer finite. RxPath proposes a circularity-test for the evaluation of such axes, such that whenever an element with the same URI reference as an anchestor is encountered that element is skipped in the evaluation. (One consequence of this approach is that blank nodes need to be assigned a unique URI reference.)

Furthermore, RxPath changes the semantics of the closure axes to only consider elements representing RDF properties in the original RDF model (this is easy as the mapping from RDF into an XML document discussed above uses a striped representation of RDF statements [73]). Furthermore, an expression such as `descendant::rdf:type` only matches an element representing an `rdf:type` property where all elements representing an RDF property actually represent an `rdf:type` property. In other words, `descendant::rdf:type` is more similar to the regular tree expression `(rdf:type)*` than to the XPath expression `descendant::rdf:type`.

Once more we use the same variation of Query 2 as above to illustrate the language syntax: "Select the names of all authors of historical essays with the title 'Bellum Civile'." (assuming the `books` prefix is bound to `http://example.org/books-rdfs#`):

```
/books:Historical_Essay[books:title = 'Bellum Civile']/
books:author/*/books:authorName
```

Based on RxPath two more languages have been defined: RxSLT [347]is "syntactically identical to XSLT 1.0" [347], but uses RxPath instead of XPath 1.0. RxUpdate [348] is syntactically very similar to XUpdate [247], but again uses RxPath instead of XPath to update RDF models.

Project page:
`http://rx4rdf.liminalzone.org/rx4rdf`

Implementation(s):
prototype in Python, available from project page

Online demonstration:
none

4.3.2.6 Versa

Conceived as query language for the Python-based 4Suite[2] toolkit for XML and RDF application development, Versa[3] is a query language for RDF inspired by XPath that can be used as a replacement of XPath for pattern matching in XSLT. Although inspired by XPath it is sufficiently different to deserve a discussion separate of the languages shown in the previous section.

The details of Versa are described in [295], [290] and [291] present gentle introductions into the language. The core design principles of Versa, as stated on `http://uche.ogbuji.net/tech/rdf/versa/`, are:

- "**Strong alignment with XML.**" As the 4Suite toolkit provides access to both XML and RDF data and technologies, the use of Versa to access RDF, e.g., when constructing an XML document with XSLT is an obvious choice. However, it has not been attempted to provide a single query language for both XML and RDF data, rather a set of query languages such as XPath and Versa are provided, each specialized for a certain data formalism.
- "**XPath-like idiom.**" The approach taken in the 4Suite toolkit, viz. not to provide an integrated query language for different data formats but rather to use a set of specialized query languages is all the more viable the more these query languages have in common. Therefore, Versa has been designed with a syntax inspired by XPath, although it arguably deviates quite notably, even more so than the query languages discussed in the previous section.
- "**Extensibility.**" Just as XPath, Versa is designed to be extensible in the same way as XPath, i.e., using externally defined extension functions. However, the current version of the specification [295] is not very clear on this point.
- "**Ease of learning.**" The authors claim that "many users have reported that they become proficient very quickly with Versa." Justifiably, they argue that the superficial similarity some of the other RDF query languages share with SQL is not actually helpful in many cases as there is considerable mismatch with regard to the data model used by the languages. Although the traversal constructs of Versa are designed to be easy to recognize and remember, they are considerable different to the traversal expressions in XPath (i.e., XPath axes and node-tests) or similar path-based query languages.
- "**Expressiveness.**" The set of query constructs provided by Versa covers a, for a path-based language to be used within, e.g., XSLT, surprisingly large set of the functionalities discussed in Section 3.2, lacking most notably means for defining views, functions, and other forms of construction. This lack can be justified to some extent by the intended use of Versa within some host language that might provide these means.

At the core of the Versa query language are the assorted traversal and filter expressions.

- *Forward traversal.* Versa allows the traversal of one or more properties starting from a list of subjects to select the objects that are reachable via the given properties. E.g., the expression `all() - books:author -> *` selects all resources that are author of another resource. The objects can also be restricted, e.g., to those containing a certain string. Here $*$ indicates a wildcard, i.e., no further restriction on the objects. Such traversal expressions can be chained.

  The following Versa query uses forward traversal operators to implement Query 1 (in the following discussion of Versa, the namespaces are assumed to be set up externally):

[2] `http://4suite.org/`

[3] `http://uche.ogbuji.net/tech/rdf/versa/`

```
distribute(type(books:Essay),
".",
"distribute(.−books:author→∗,
".", ".-books:authorName→∗)")
```

The `distribute()` function returns a list of lists containing the result of the second, third, … argument evaluated starting from each of the resources selected by the first argument. As in XPath, `.` denotes the current node in such a context. Here, the first argument selects all resources classified as `books:Essay` and evaluates starting with these resources the remaining two arguments: The former returns all those books, the second uses `distribute()` again to select the authors together with their name.

- *Forward filter.* Just like a traversal, where the object of the traversed statement is select, one can use a forward filter to select the subject of a statement. E.g., the Versa query `type(books:Essay) |- books:title -> eq("Bellum Gallicum")` selects all essays with a title "Bellum Civile".

- *Backward traversal.* Sometimes, one would like to navigate from the objects of a statement to the subjects. Therefore, Versa offers also a backward traversal (although, so far no backward filter is provided, it can however, be implemented with the general filter expression discussed below). E.g., the query above selecting all essays with a title "Bellum Gallicum" can also be written as

  ```
  (books:Essay <- rdf:type - ∗) |- books:title →eq("Bellum Gallicum")
  ```

- *General traversal.* Whereas the traversal operators discussed so far only allow the traversals of paths with fixed length, Versa also offers a function for general traversals, both forward and backward. This function, called `traverse`, can also be used to traverse paths of arbitrary length. E.g., the following query obtains all sub-classes of `books:Writing`:

  ```
  traverse(books:Writing, rdf:subClassOf, vtrav:inverse, vtrav:transitive)
  ```

- *General filter.* Similarly, the `filter` function provides a general filter, where the result of evaluating the first argument is filtered by the remaining arguments evaluated in the context of the elements selected by the first one. E.g., to select all essays with title "Bellum Gallicum" and a translator with name "J. M. Carter" could be implemented by the following query:

  ```
  filter(books:Essay <- rdf:type - ∗,
  ". − books:title →eq('Bellum Gallicum')",
  ". − books:translator →books:translatorName →eq('J. M. Carter')"
  ```

Following this short overview over the core traversal and filtering functions of Versa, the language is further investigated following the criteria proposed in Section 3.2.

Easy of use. Although designed to be closely aligned with XPath, the traversal operators shown above and the use of functions instead of specific syntactical constructs such as predicates in XPath gives Versa a rather unfamiliar feeling compared to other XPath-oriented languages.

Functionality—Query Types. Selection and extraction queries can be easily implemented in Versa, although as demonstrated by the above implementation of Query 1, the selection of multiple related items is not very convenient. In contrast to most other RDF query languages, Versa allows the extraction of arbitrary size graphs, as required by Query 2. Also reduction queries can be expressed, e.g., using negation or set difference. Query 3 can be implemented in Versa by the following query

```
difference(all(),
union(type(rdfs:Class),
union(type(rdfs:Property,
all() <- books:translator - *))
)
)
```

This query selects all resources except for those that are either (a) a `rdfs:Class`, (b) a `rdfs:Property`, or (c) occur as object in a statement with predicate `books:translator`.

Neither restructuring nor combination or inference queries can be directly expressed in Versa, as the result of a Versa query is always a list (or possibly a list of lists). However, queries such as Query 4 and 9 can be approximated, e.g., by returning all the tuples of co-authors of a book:

```
distribute(all(), ". − books:author →∗", ". − books:author →∗")
```

However, this query will also include that, e.g., "Julius Caesar" is a co-author of himself. It does not seem possible to avoid this in Versa, as the later arguments of `distribute` are evaluated independently.

Versa also provides a large set of aggregation functions. Query 5 can be implemented by the following program

```
max(filter(all(),
". − books:author →books:authorName →eq(’Julius Caesar’)"
)
- books:year →∗)
```

Starting from the filtered books, the year of publication of those books is selected and the maximum of these years calculated.

Also Query 6 can be implemented in Versa using the `length` function for calculating the number of authors per book:

```
distribute(traverse(books:Writing, rdf:subClassOf,
vtrav:inverse,vtrav:transitive),
".",
"max(length((. <− rdf:type ∗) − books:author →∗))"
)
```

Semantics. No formal semantics has been provided so far.

Complexity and implementation. No formal complexity study of the language has been published so far. Only a single implementations by the authors of the language is available.

Reasoning. No reasoning abilities are provided.

Ontology awareness. Versa provides a RDFS-aware `type()` function that returns all resources that are classified under the given RDFS class or one of its sub-classes. The transitive semantics of `rdfs:subClassOf` and `rdfs:subPropertyOf` are not provided by default but can be implemented by the general traversal function `traverse`.

Project page:
`http://uche.ogbuji.net/tech/rdf/versa/`

Implementation(s):
available as part of 4Suite from `http://4suite.org/`

Online demonstration:
none

4.3.3 RQL-family

4.3.3.1 RQL

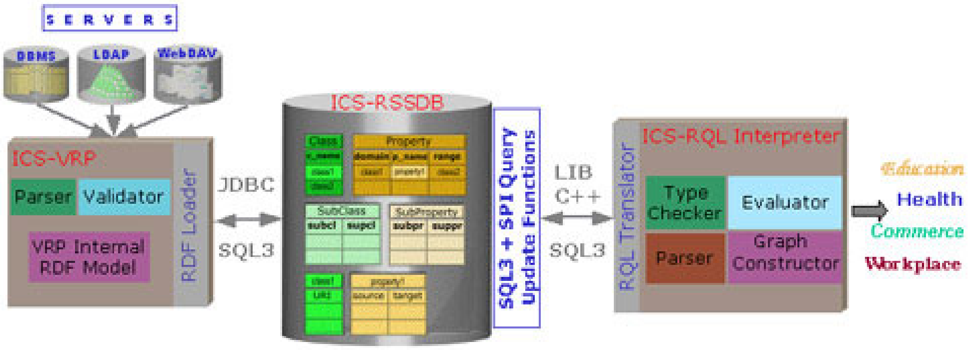
RQL [119, 219, 218, 220, 221], the RDF Query Language, has been developed at the ICS-FORTH and influenced several later proposals for RDF query languages such as SeRQL (cf. Section 4.3.3.2, eRQL (cf. Section 4.3.3.3, and BRQL (cf. Section 4.3.1.4). The reference implementation of RQL has been developed as part of the RDFSuite [10], a collection of tools that provide efficient access to increasingly large RDF stores drawing on established database technology provided by an ORDBMS. The architecture of RDFSuite is shown in Figure 4.1 identifying the three main components of RDFSuite:

- the Validating RDF Parser (VRP), a high-performance, RDFS-aware RDF parser that also allows the user to specify semantic constraints against which a document should be validated,

- the RDF Schema-Specific Data Base (RSSDB), a persistent RDF store based upon the ORDBMS PostgreSQL (`http://www.postgresql.org`) that stores RDF data based on its (RDFS) schema, and

- an interpreter for RQL, implemented on top of the RSSDB.

Figure 4.1 RDFSuite Architecture
(from [9], © ERCIM News)



RQL has also been used in the ICS-FORTH Semantic Web Integration Middleware (SWIM) [118], where it is used to query data represented as RDF but integrated from different data sources, e.g., XML documents or relational databases.

In contrast to RDF query languages such as RDQL, that are tailored to be easy to use by providing only a small set of (often used) query constructs excluding, e.g., the direct exploitation of RDFS or direct support for more complex aspects of RDF such as containers or reification, RQL also has the ability to combine schema and data querying and allows the specification of complex graph patterns.

At the core of RQL is a formal data model for RDF graph data (deviating slightly from standard RDF/S semantics by disallowing cycles in the subsumption hierarchy and requiring that for each property both domain and range are defined for (a) simplification and (b) alignment with underlying type systems) based upon typing information provided by an RDFS schema. The salient features of this data model are

- a clear separation between the different RDF/S abstraction layers: (1) data, i.e., description of resources such as persons, XML documents, etc., (2) schemas, i.e., classifications for such resources, and (3) metaschemas containing the metaclasses (i.e., classes of classes or properties) `rdfs:Class` and `rdfs:Property` and their refinements)

- flexible type system tailored to the specificities of RDF/S by allowing (a) optional and multi-valued properties, (b) superimposed descriptions of the same resources (i.e., resources can be classified under different classes not related in the subsumption hierarchy) and (c) the flexible refinement of schemas.

Based upon this data model, RQL provide a number of novel query constructs for querying the type information associated with the RDF data. In the following a condensed overview of the most prominent query constructs provided in RQL given, more details can be found in [119, 219, 218, 220, 221].

Basic schema queries. As stated above, one of the salient features of RQL is the use of type information derived from an RDFS schema. The subsumption hierarchy defined by `rdfs:subClassOf` and `rdfs:subPropertyOf` can be accessed in different manners:

- Querying the *sub-classes* of a class: E.g., `subClassOf(books:Writing)` returns all sub-classes of `books:Writing` (assuming the namespace `books` is set up properly using `USING NAMESPACE books = &http://example.org/books-rdfs#`).

- Querying the *sub-properties* of a property using `subPropertyOf`.

- Querying the **domain and range of a property**. E.g., the following query obtains instances of which classes can be combined with `books:author`:

  ```
  SELECT $C1, $C2
  FROM {$C1}books:author{$C2}
  USING NAMESPACE books = &http://example.org/books#
  ```

  The `$` prefix of a variable indicates a “class variable”, i.e., a variable ranging on schema classes, in other words, resources with `rdf:type rdfs:Class`. Therefore, `$C1` selects all classes that are in the domain of `books:author` and `$C2` those in its range.

  More explicitly, the query can be stated as (however, since subClassOf is not reflexive, the direct domain and range are left out):

```
SELECT C1, C2
FROM subClassOf(domain(book:author)){C1}, subClassOf(range(books:author)){C2}
USING NAMESPACE books = &http://example.org/books#
```

Note, that here the query variables are *not* prefixed by `$`, therefore the values (and not their type) returned by the `subClassOf` expressions are used. This is the intended result, as the values returned by `subClassOf` are already the classes searched for.

Finally, the query could also be formulated using a type constraint:

```
SELECT C1, C2
FROM Class{C1}, Class{C2}, {;C1}books:author{;C2}
USING NAMESPACE books = &http://example.org/books#
```

The first part of the `FROM` clause (`ClassC1, ClassC2`) selects all pairs of classes defined in the schema. These pairs are further constrained by the second part, that stipulates that only such pairs are to be retained where the first class can occur as domain and the second as range of `books:author`. This restriction is expressed by a type constraint, written (in general) `{X;Y}`, where `X` is bound to the concrete values (here, e.g., "Julius Caesar") and `Y` to the type of the value (here, e.g., `Literal`, i.e., the class of all RDF literals.

- Querying **tops and leafs** of the subsumption hierarchy. E.g., `topclass(books:Historical_Essay)` returns `books:Writing` on the data from Figure 2.1. Furthermore, it is possible to query the nearest common ancestor of two classes, e.g., `nca(books:Historical_Essay, books:Historical_Novel)` is `books:Essay`.

- Querying **meta-schema** information. Meta-classes, such as `Class`, and `Property` (the class of all classes, all RDF literals, all properties, resp.) can be queried just as any other class. Obviously, the extent of a meta-class is a set of classes, as demonstrated above.

- Querying RDF **properties**. Just as classes can be queried using class variables indicated by a `$`-prefix, RDF properties are selected by "property variables" prefixed by `@`. E.g., the following query selects all properties together with their range that can be attached to resources classified as `books:Writing`:

```
SELECT @P, $V
FROM {;books:Writing}@P{$V}
USING NAMESPACE books = &http://example.org/books#
```

Combining these facilities, one can easily implement Query 8 in RQL:

```
SELECT X, Y
FROM Class{X}, subClassOf(X){Y}
```

This query returns pairs of all classes such that the first class is a super-class of the second one.

Data queries. Obviously, RQL also provides access to the actual resource descriptions. One can access resources by their type and by navigating to their position in the RDF graph, and further restrictions to the data to be selected can be specified by filters:

- Querying the *extent* of classes and properties. Both classes and properties can be queried for their (direct and indirect) extent:

```
books:Writing
```

returns all resources classified as `books:Writing` or one of its sub-classes. This is equivalent to

```
SELECT X
FROM books:Writing{X}
```

If only those resources X shall be returned that are directly classified as `books:Writing`, i.e., where an RDF tuple $(X, \texttt{rdf:type}, \texttt{books:Writing})$ exists, the name of the class must be prefixed by `^`.

Similarly, one can query the extent of a property, e.g.,

```
^books:author
```

returns pairs of all resources X, Y that stand in the `books:author` relation, i.e., where an RDF tuple $(X, \texttt{books:author}, Y)$ exists.

- Querying by navigation using **generalized path epressions** [117]. RQL uses generalized path expressions known from, e.g., OQL and Lorel, to allow navigation both in the data and in the schema graph.

  This allows for an easy traversal of relations convenient for implementing Query 1:

  ```
  SELECT X, Y, Z
  FROM {X;books:Essay}books:author{Y}.books:authorName{Z}
  USING NAMESPACE books = &http://example.org/books#
  ```

- **Filtering** the result. The result of a query can be further restricted by a `WHERE` clause. E.g., to select only books and the names of their authors if the title of the book is `"Bellum Civile"`, one could add a `WHERE` clause as in the following query:

  ```
  SELECT X, Y
  FROM {X;books:Essay}books:author.books:authorName{Y}, {X}books:title{T}
  WHERE T = "Bellum Civile"
  USING NAMESPACE books = &http://example.org/books#
  ```

Mixing schema and data queries. Querying data and schema can be intertwined in any way. In particular, as shown above, data can be selected based on its type, by providing filter expressions for the variable selecting the data. E.g., the expression `X;books:Essay` restricts bindings for variable `X` to resources with type `books:Essay`.

Often interesting queries not only require the use of type information for filtering, but also benefit from providing type information in the result. Query 2, that asks for a kind of description of the book with title "Bellum Civile", could be implemented in RQL in the following way (interpreting "description" as the schema of a resource):

```
SELECT $C, (SELECT @P, Y
FROM {Z ; ^$D} ^@P {Y}
WHERE Z = X and $D = $C)
FROM ^$C {X}, {X}books:title{T}
WHERE T = "Bellum Civile"
USING NAMESPACE books = &http://example.org/books#
```

This query returns all classes under which the resource with title "Bellum Civile" is directly classified (`^$C{X}` selects all values in the direct extent of any class). Grouped by these classes, all properties and classes that can be used as values for the respective property are queried by a nested RQL query and returned.

Several of the features of RQL are not discussed in length here, such as support for containers, aggregation, schema discovery, etc.

Although the original proposal of RQL does not include view definition constructs, RVL [254] provides such an extension. Using this language, one could, e.g., define the inverse relation of `books:author` as a view by the following program:

```
CREATE NAMESPACE mybooks = &http://example.org/books-rdfs-extension#
VIEW authored(Y, X)
FROM {X}books:author{Y}
USING NAMESPACE books = &http://example.org/books#
```

Such a view can be queried just like any other data, e.g., to select all authors that authored a book translated by someone called "Carter" together with these books:

```
SELECT X, Y
FROM {X}mybooks:authored{Y}books:translator{T}
WHERE T like "*Carter*"
USING NAMESPACE books = &http://example.org/books#
USING NAMESPACE mybooks = &http://example.org/books-rdfs-extension#
```

Concluding this overview of RQL is a short discussion of the query language under the evaluation criteria proposed in Chapter 3.

Easy of use. The comparatively large and diverse set of features offered by RQL, naturally lead to a somewhat more complicated syntax and semantics if compared to basic query languages such as RDQL (cf. Section 4.3.1.3. One critique voiced when considering RQL is that the typing features of the language, although useful for some scenarios, actually complicate the expression of simple queries and are hard to implement. Therefore, both of the proposals for simplifying RQL, viz. SeRQL and eRQL, do not provide the same level of type information as RQL.

Also the use (and representation) of a number of syntactic constructs, e.g., for distinguishing class (using `$`-prefix), property (using `@`-prefix), and data variables (using no prefix) or for separating direct (`^`-prefix) and full extent (no prefix) of a class, is neither intuitive nor based on established conventions. Providing also a more verbose version of such constructs, might improve readability and ease the familiarization with the language.

Functionality—Query Types. As stated already, RQL is far more expressive than basic RDF query languages such as the SquishQL family. Actually, most of the queries discussed in Section 3.2 can be easily expressed in RQL (here, we also include the view definitions provided by RVL) with the notable exception of those queries requiring means for traversing the transitive closure of arbitrary relations, instead of only the two relations `rdfs:subClassOf` and `rdfs:subPropertyOf` that constitute the subsumption hierarchy.

An implementation of Query 1 has already been given above. Query 2 can not be expressed in RQL exactly, as there is no means to select "everything related to some resource". However, in the above discussion a modified version of this query, where a resource is described by its schema, is shown. Reduction queries such as Query 3 can often concisely be expressed in RQL, in particular, if the reduction query is, as in this case, based on type information:

```
SELECT S, @P, O
FROM (Resources minus (SELECT T FROM {B}books:translator{T})){S},
(Resources minus (SELECT T FROM {B}books:translator{T})){S},
{S}@P{O}
USING NAMESPACE books = &http://example.org/books#
```

This query returns all triples where both the subject and the predicate of the triple is some Resource that does not occur as object in a statement with predicate `books:translator`. Note, that `Resource` is the top-level *schema* class, but does not include resources classified as `rdfs:Class` or `rdfs:Property` as these are considered schema objects and therefore contained in a meta-schema class.

An implementation of Query 4 is given above using RVL.

Aggregation queries are also supported by RQL, e.g., consider Query 5 and a possible implementation in RQL shown below:

```
max(SELECT Y
FROM {B;books:Writing}books:author.books:authorName{A},
{B}books:pubYear{Y}
WHERE A = "Julius Caesar")
```

Inference and combination queries often require traversing the transitive closure of arbitrary relations or even general recursion, that can both not be expressed in RQL. However, when the inference does not need recursion, as in Query 9, the query can be expressed in RQL

```
SELECT A1, A2
FROM {Z}books:author{A1}, {Z}books:author{A2}
WHERE A1 != A2
USING NAMESPACE books = &http://example.org/books#
```

or in RVL (allowing the results to be queried as any relation in the original data)

```
CREATE NAMESPACE mybooks = &http://example.org/books-rdfs-extension#
VIEW mybooks:co-author(A1, A2)
FROM {Z}books:author{A1}, {Z}books:author{A2}
WHERE A1 != A2
USING NAMESPACE books = &http://example.org/books#
```

Semantics. RQL has been designed around a formal data model for RDF. Based upon this data model, both typing rules and formal interpretations of RQL queries have been specified [221]. As discussed above, the data model slightly deviates from a standard RDF interpretation (as specified in [246]).

Complexity and implementation. No formal complexity study of the language has been published so far. Aside of the implementation provided by the authors of the language as part of the ICS-FORTH RDFSuite, the only other implementation of RQL the authors of this survey are aware of is as part of Sesame [79]. However, for Sesame the typing features of RQL have been disregarded for the most part and a new RDF query language, called SeRQL, that is based upon RQL has been specified. It is discussed in the following section.

Reasoning. No specific reasoning abilities are provided. In particular, the only form of recursion is the ability to navigate the transitive closure of certain predefined schema relations. However, some non-recursive inference queries can be realized as shown above. In particular, RQL supports a rich set of boolean expressions including negation and quantification.

Ontology awareness. RQL strongly relies on RDFS for providing type information on the data to be queried. (An extension to) OWL has not been considered so far.

Project page:
`http://139.91.183.30:9090/RDF/RQL/`

Implementation(s):
RDFSuite (`http://139.91.183.30:9090/RDF/index.html`), Sesame (`http://www.openrdf.org/`) without type system

Online demonstration:
`http://139.91.183.30:8999/RQLdemo/`, based on Sesame: `http://www.openrdf.org/demo.jsp`

4.3.3.2 SeRQL

As part of the EU research project On-To-Knowledge (`http://www.ontoknowledge.org/`), the Sesame [79] RDF database has been developed. An initial review of query languages for semi-structured data and RDF [76] suggested the use of RQL as query language for the On-To-Knowledge project. In [77], this choice is detailed by defining the query language to be used in the On-To-Knowledge project, tentatively called OTK-RQL. This language is still syntactically nearly identical to RQL, but leaves out in particular RQL's entire type system. The current version of Sesame still supports RQL (without typing of queries), but also provides a novel RDF/RDFS query language claiming to "combine the best features of other (query) languages (RQL, RDQL, N-Triples, N3)" [95] into a "second generation RDF query language" [78].

SeRQL is described in detail in [95] and [78]. Here, we focus on the differences to previous RDF query languages, in particular to RQL.

The most striking differences between RQL and SeRQL are:

- SeRQL does not provide any form of **typing** for RDF resources, only typed literals as provided by the recent revision of RDF [288] are considered. One reasons the others of SeRQL offer for avoiding to provide typing in the query language (aside of the increase complexity of both implementation and language) is that RDF schema languages are designed to be extensible and a typed query language therefore needs to provide some mechanism for integrating such extensions (as provided, e.g., by OWL), making an implementation of such a query language even harder.

- In an effort to simplify common queries, SeRQL modifies and extends the general **path expressions** used in RQL: Basic path expressions use a syntax similar to RQL, e.g., `{X} <rdf:type> {<books:Essay>}` is the SeRQL notation for a path expression that returns all resources classified under `books:Essay` as bindings for the variable `X`. Compound path expressions, e.g., for selecting a book together with the names of its authors, use an "empty node", denoted as `{}`, instead of the path concatenation `.` commonly used for combining path expressions. The RQL path expression `{X}books:author.books:authorName{Y}` becomes in SeRQL `{X}<books:author>{}<books:authorName>{Y}`. Furthermore, a number of short cuts addressing common queries are provided:

  – Querying *multi-valued properties*. Properties in RDF can have multiple values and SeRQL provides a short cut to query several of these values in a single path expression. This allows for an easy formulation of Query 9 (note, the use of `<!` and `>` for enclosing a URI in contrast to the use of simple angle brackets (without the exclamation mark) for enclosing QNames)

```
CONSTRUCT {X} <mybooks:co-author> {Y}
FROM {Book} <books:author> {X, Y}
WHERE X != Y
```

```
USING NAMESPACE books = <!http://example.org/books#>
mybooks = <!http://example.org/books-rdfs-extension#>
```

This query is equivalent to

```
CONSTRUCT {X} <mybooks:co-author> {Y}
FROM {Book} <books:author> {X},
{Book} <books:author> {Y}
WHERE X != Y
USING NAMESPACE books = <!http://example.org/books#>
mybooks = <!http://example.org/books-rdfs-extension#>
```

  - Querying *multiple properties* of the same resource. Often one is interested not only in a single, but rather in multiple properties of a resource. The following query illustrates the syntactic short hand provided by SeRQL for this case: It selects the authors of all books with title "Bellum Civile" and a translator with name "J. M. Carter".

    ```
    SELECT Author
    FROM {Book} <books:title> {"Bellum Gallicum"};
    <books:translator> {} <books:translatorName> {"J. M. Carter"};
    <books:author> {Author}
    USING NAMESPACE books = <!http://example.org/books#>
    ```

    Note, the use of the semicolon for separating expressions with the same subject.

  - Querying *reified statements*. A reified expression can be queried by explicitly asking for the four triples associating the statement with its type (`rdf:Statement`), subject, predicate, and object. As this is often bothersome, SeRQL allows reified statements to be queried by enclosing the non-reified version of the statement in curly brackets.

- As all properties in RDF are essentially **optional** (unless considering schema languages such as OWL), queries where some information is retrieved if it is available without making the query fail if the information misses are a natural requirement for an RDF query language. SeRQL provides optional path expressions by enclosing an arbitrary path expressions (including those using the short hand constructs described above) in square brackets. E.g., the following query retrieves books together with their title and optionally their translators. If there is a translator and an age for that translator is specified, that information should also be returned.

  ```
  SELECT *
  FROM {Book} <books:title> {Title};
  [<books:translator> {Translator}
  [ <books:age> {Age} ] ]
  USING NAMESPACE books = <!http://example.org/books#>
  ```

  Note, the nesting of the optional expressions and the use of * to include bindings for all variables from the query in the result.

Aside of these issues, it should be noted that the access to schema information has been reduced to the provision of both the original and the intransitive view of the two RDFS relations defining the subsumption hierarchy.

Easy of use. The expressed goal of the SeRQL authors has been a simplification of previous approaches such as RQL while retaining the, from the point of view of the SeRQL authors, most useful features that distinguish RQL from basic RDF query languages such as RDQL. Arguably, simply by reducing the number of language constructs, SeRQL is easier to grasp and use than RQL. Also, the syntactic short hands are, where possible, aligned with constructs from established RDF syntaxes such as N3 or N-Triples.

However, this is obviously paid for by reducing the expressiveness of the language in comparison to RQL, therefore requiring more effort for writing complex queries that could benefit from the more expressive features of RQL.

Functionality—Query Types. As expected, SeRQL can *not* express all of the queries from Section 3.2 that could be expressed in RQL but still provides more functionality than RDQL. Selection and extraction queries can be easily expressed in SeRQL with the same limitation as in the case of RQL, viz. that it is not possible to navigate arbitrary length paths in the graph, e.g., for returning all statements related to a resource or a "concise bounded description" as defined in [351].

In contrast to RQL, SeRQL currently neither provides set operations nor existential or all quantifiers. Therefore, Query 3 can not be expressed in SeRQL.

Thanks to the ability to construct new statements using the `CONSTRUCT` clause, SeRQL can express restructuring and simple inference queries as shown above. The restructuring Query 4 can be expressed as

```
CONSTRUCT {Author} <mybooks:authored> {Book}
FROM {Book} <books:author> {Author}
USING NAMESPACE books = <!http://example.org/books#>
mybooks = <!http://example.org/books-rdfs-extension#>
```

Aggregation queries can not be expressed, although [95] states that the addition of aggregation queries to SeRQL is planned.

As shown above, some simple inference queries such as Query 9 can actually be implemented in SeRQL and thanks to the RDFS-aware storage in Sesame the transitive closure of `rdfs:subClassOf` is provided in SeRQL. However, neither the transitive closure of arbitrary relations nor general recursion can be expressed.

Semantics. No formal semantics has been provided so far, however a formal algebraic model is being planned according to [95].

Complexity and implementation. No formal complexity study of the language has been published so far. There are currently two independent implementations of the language.

Reasoning. Again, only limited reasoning abilities are provided. Using the `CONSTRUCT` clause one can implement basic derivations as demonstrated above, however no recursion is provided.

Ontology awareness. SeRQL is RDFS aware in that it provides the ability to query both the explicitly stored subsumption relations and their transitive closure.

Project page:

Sesame `http://www.openrdf.org/`

Implementation(s):

available from the Sesame project page, an implementation in Prolog using the SWI-Prolog[4] Semantic Web library is provided at `http://gollem.swi.psy.uva.nl/twiki/pl/bin/view/Library/SeRQL`

[4] `http://www.swi-prolog.org/`

Online demonstration:

several ones (featuring not only SeRQL as query language, but also RDQL and RQL) accessible at `http://www.openrdf.org/demo.jsp`

4.3.3.3 eRQL

In contrast to SeRQL, which aims at providing a language more balanced between expressiveness and ease-of-use than RQL, eRQL [358] proposes a radical simplification using essentially a keyword-based interface similar to popular information retrieval systems. It is the expressed goal of the eRQL authors, to provide a "*Google-like* query language but also with the capacity to profit of the additional information given by the RDF data"[5].

eRQL only provides three query constructs:

- *One-word queries.* Single keywords are valid eRQL queries. E.g., the query `CAESAR` returns all statements such that the string "CAESAR" occurs in their URI or literal value of the subject, predicate, or object of the statement using case-insensitive matching. Surprisingly, phrase queries (e.g., the phrase "Bellum civile") do not seem to be expressible in eRQL.

- *Neighborhood queries.* Instead of returning only the statements directly containing a keyword as in the first case, neighborhood queries allow the user to also select all statements related to (i.e., "in the neighborhood of") a such a statement. E.g., the query `{{CAESAR}}` returns all statements connected by at most two edges in the RDF graph to a node containing "CAESAR". On the data from Figure 2.1, the following triples are returned:

  ```
  _:1 books:author _:2.
  _:1 books:authorName "Julius Caesar".
  _:1 books:author _:3.
  _:1 books:title ‘‘Bellum Civile’’.
  _:1 books:translator _:4.
  ```

  Using `{{{CAESAR}}}` would also include the names of the other others and the translator, as well as the triple classifying the book as `books:Historical_Essay`. eRQL allows any (finite) number of such brackets to select a neighborhood of the specified size around the base triples.

- *Conjunctive and disjunctive queries.* Both, neighborhood and one-word queries can be combined using the boolean operators `AND` and `OR`. No negation is provided, however.

eRQL does not allow the expression of most of the queries given in Section 3.2, since the abilities of eRQL are more akin to an information retrieval language than a conventional query language. However, to some extent Query 2 can be expressed simply by the query

```
{{"Bellum" AND "Civile"}}
```

This query returns all statements containing both the string "Bellum" and the string "Civile" in the URI or literal value of the subject, predicate, or object together with all statements reachable from these within two steps. However, this is only a vague approximation of the actual intent of the query. In particular, eRQL does not allow the selection of a neighborhood with previously unknown size around a resource (e.g., for obtaining a "concise-bounded descriptions" [351]). In contrast to the claims of the authors of eRQL, such limited neighborhood queries require a-priori knowledge of the schema of the data to be queried.

[5] `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`

Nevertheless, eRQL is one of the few approaches aiming at a combination of information retrieval features and RDF querying, the need for which is evident when considering the use of RDF for improving searching in the (Semantic) Web.

Project page:
`http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`

Implementation(s):
eRQLEngine, available from `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`

Online demonstration:
none

4.3.4 Query Languages using a Controlled Natural Language

4.3.4.1 Metalog

Metalog [261, 262, 260] is a system for querying and reasoning with Semantic Web data. It has been introduced around 1998 in [261] leading to the claim of the project page that "Metalog has been the first semantic web system to be designed, introducing reasoning within the Semantic Web infrastructure by adding the query/logical layer on top of RDF" [`http://www.w3.org/RDF/Metalog/`].

For two reasons, Metalog is notably different from most of the other RDF query languages discussed in this chapter:

- It combines querying with *reasoning* abilities such as implications and explicit representation of negative information.

- The language syntax is similar to a restricted form of natural language, where only certain keywords (and their order) is relevant and all other words are discarded. This allows for easy understanding of queries, although authoring still requires knowledge about the exact keywords and how to use them.

Metalog's syntax uses (English) sentences to describe a query. Within a sentence only three kinds of tokens are actually meaningful for the Metalog processor: variables (or "representations") are written in capital letters only, any quote-delimited string is recognized as a "name" and interpreted either as an RDF literal or a URI, and keywords connect variables and names to queries. The list of keywords available includes of statements, logical implication expressed by `then`, `imply`, or `implies`, definition of variables using `represents`, `order` for building RDF sequence containers, and various keywords for arithmetic expressions.

Metalog also provides a natural-language-like syntax for stating RDF triples.

The following Metalog program implements Query 1:

```
comment: some definitions of variables (or representations)
ESSAY represents the term "Essay"
from the ontology "http://example.org/books#".
AUTHORED-BY represents the verb "author"
from the ontology "http://example.org/books#".
IS represents the verb "rdf:type"
from RDF "http://www.w3.org/1999/02/22−rdf−syntax−ns#".
BELLUM_CIVILE represents the book "Bellum_Civile"
from the collection of books "http://example.org/books#".
comment: RDF triples written as Metalog statements.
BELLUM_CIVILE IS an ESSAY.
BELLUM_CIVILE is AUTHORED-BY "Julius Caesar".
BELLUM_CIVILE is AUTHORED-BY "Aulus Hirtius".
comment: a Metalog query
do you know SOMETHING that IS an ESSAY and that is AUTHORED-BY SOMEONE?
```

As answer to the query shown in the last line of the Metalog program, the interpreter answers with all definitions and the first two RDF triples as first result, the first and the last RDF triple as second result.

An interesting observation about Metalog can be drawn from [259]: A natural-language layer on top of the textual or XML syntax of any RDF query language might help, in particular non-experts, to quickly grasp the meaning of queries and what result to expect. In this sense, a translation of Metalog or a similar restricted form of natural language into some of the more traditional and, in many cases, better performing approaches discussed in this survey might be worth investigating.

Project page:
`http://www.w3.org/RDF/Metalog/`

Implementation(s):
prototype, available from the project page

Online demonstration:
none

4.3.5 Others

4.3.5.1 Algae

Algae[6] [310, 312] is an RDF query language developed as part of the W3C Annotea project (`http://www.w3.org/2001/Annotea/`). The Annotea project provides a research platform for novel collaborative applications based on shared metadata such as Web annotations, bookmarks, comments, explanations, etc. When users access a Web site with an Annotea-enabled browser (such as Amaya (`http://www.w3.org/Amaya/`), Mozilla or Internet Explorer, the latter two require extensions for using Annotea), one or several *annotation servers* are contacted to deliver annotations for the currently visited Web site or to provide a classification of that Web site based on previously stored bookmarks.

The efficient retrieval of the related information from the annotation server is clearly of high relevance in such a setting. As the annotations are stored in RDF, an RDF query language, called Algae, has been developed to address the special needs of this application including the need for updates and simple reactive rules. In [312], Algae is described in more detail and proposed as a general-purpose RDF query language.

Algae is centered around two concepts:

- "Actions" are the directives `ask`, `assert`, and `fwrule` that determine whether an expression is used to query the RDF data, insert data into the graph, or to specify ECA-like rules. Only the first of these is mandatory for an Algae processor, the two others are defined in an extension module described in [311].

- Algae queries produce result sets containing not only bindings for query variables (as, e.g., RDQL [332]) does, but also triples from the RDF graph that constitute "proofs" for the solution, i.e., that are required to justify that a certain combination of bindings for the query variables is actually a match for the query. It is possible to combine several sub-queries in a single Algae expression in which case the results of each sub-query are combined into a single result set.

[6] The current version is sometimes also referred to as Algae2, since there has been an earlier incarnation with more limited querying abilities. In this survey, we follow [312] in referring to the language as simply "Algae".

Table 4.1 Algae result set

?title	?translator	*Proof*
“Bellum Civile”	“J. M. Carter”	_:1 rdf:type <http://exam...ks-rdfs#Essay>. _:1 books:author _:2. _:2 books:authorName ‘‘Julius Caesar’’. _:1 books:title ‘‘Bellum Civile’’. _:1 books:translator ‘‘J. M. Carter’’.

Syntactically Algae is based on N-triples (described in [186]) for representing and querying RDF triples. This simple triple syntax is extended by the above mentioned action directives and so-called “constraints”, written between curly brackets, that specify further arithmetic or string comparisons that must be fulfilled by a selected tuple.

To illustrate the abilities of Algae, consider Query 1. This query could be realized by the following Algae expression:

```
ns rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ns books = <http://example.org/books#>
read <http://example.org/books> ()
ask ( ?essay rdf:type <http://example.org/books#Essay> .
?essay books:author ?author .
?author books:authorName ?authorName )
collect( ?essay, ?author, ?authorName )
```

This query becomes more interesting, if we are only interested in the titles of essays written by “Julius Caesar” but also want the translators of such books returned, if there are any:

```
ns rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ns books = <http://example.org/books#>
read <http://example.org/books> ()
ask ( ?essay rdf:type <http://example.org/books#Essay> .
?essay books:author ?author .
?author books:authorName ‘‘Julius Caesar’’ .
?essay books:title ?title .
~?essay books:translator ?translator .
)
collect( ?title, ?translatorName )
```

Note the use of ~ to declare the translator an optional triple in the query.

Such a query executed against the sample data shown in Figure 2.1, would return the following result set shown in Table 4.1.

Easy of use Although neither a visual syntax nor a natural language interface is provided, Algae queries are easy to write and comprehend, although the syntax does not reminiscent of any particular other query language. In particular, the choice of N-triples (described in [186]) for representing triples is helpful for easy query formulation. Some of the syntactic short hands, such as ~ require some getting used to, but offer the advantage of a concise query formulation.

Algae offers an extension mechanism that allows the basic functionality of the language, i.e. the `ask` action directive, to be separated from more advanced features such as updates and rules (with the action directives `assert` and `fwrule`) and sorting. A query can state which of the language extensions are required to evaluate a query.

Functionality—Query Types From the queries discussed in Section 3.2, Query 2 can not be expressed due to the lack of some form of closure or recursion. For the same reasons and additionally the lack of negation, Query 4 can not be expressed. The support for Queries 5 and 6 falls short due to the lack of aggregation operators. All other queries can be expressed in Algae, albeit most of them require the extended action directives discussed in [311].

For Query 4 one could use the following Algae query:

```
ns books = <http://example.org/books#>
read <http://example.org/books> ()
fwrule ask( ?book books:author ?author )
assert ( ?author books:authored ?book )
```

Note, however that this query actually becomes an inference rule, as there are no means described to retract the old `books:author` relation.

Hence, this realization of Query 4 is more akin to a inference query such as Query 9 that could actually be implemented as

```
ns books = <http://example.org/books#>
read <http://example.org/books> ()
fwrule ask ( ?book books:author ?author1 .
?book books:author ?author2 { ?author1 != ?author2 }
)
assert( ?author1 books:co-author ?author2 )
```

Semantics No formal semantics has been published for Algae.

Complexity and implementation Algae has been implemented in the W3C Annotation Server as part of the Annotea Project. The data can be stored in-memory, in a relational database, as described in [313], or directly generated from application data. As discussed above, Algae provides extensions for updating and

No formal complexity study of the language has been published so far.

Reasoning In [311], an extension for Algae is described that allows the statement of rules for intensional data specification: If a given query succeeds, new data is added to the data store possible drawing data from the result of the query. These rules are more similar to ECA-rules, albeit with an action part limited to inserting new data, than to view definitions. No further description of reasoning mechanisms in Algae is provided.

Ontology awareness As already specified, the language ignores every kind of ontological aspect, including typing mechanisms. If such aspects must be considered, they must be treated like all the user-defined data.

It is, however, possible to implement the special semantics of RDFS relations such as `rdfs:subClassOf` using Algae rule notation. The following is an implementation of Query 8 in Algae, in that it adds the transitive closure over `rdfs:subClassOf` to the data store.

```
ns rdfs = <http://www.w3.org/2000/01/rdf-schema#>
read <http://example.org/books> ()
fwrule ask ( ?X rdfs:subClassOf ?Z.
?Z rdfs:subClassOf ?Y
)
assert ( ?X rdfs:subClassOf ?Y )
```

Project page:

http://www.w3.org/2004/05/06-Algae/ and for the Annotea project http://www.w3.org/2001/Annotea/

Implementation(s):

W3C Annotation Server http://annotest.w3.org/annotations

Online demonstration:

Query interface to the W3C Annotation Server using Algae as query language: http://annotest.w3.org/annotations?explain=false

4.3.5.2 iTQL

For the Kowari Metastore, an open source, scalable, transaction safe database for the storage of metadata, an RDF query language called iTQL [1] has been defined. iTQL provides commands not only for querying (`select`), but also for updates (`delete`, `insert`) and transaction management (`commit`, `rollback`). The syntax of iTQL is similar to SQL and therefore also reminds of RDQL. As for RDQL, the querying abilities of the language are rather limited, mostly simple selection is supported.

To illustrate the abilities of the query language, consider again Query 1 and a possible realization in iTQL:

```
alias <http://example.org/books#> as books;
alias <http://www.w3.org/2000/01/rdf-schema#> as rdfs;
alias <http://www.w3.org/1999/02/22-rdf-syntax-ns#> as rdf;
select $essay, $author, $authorName
where $essay <books:author> $author
and $author <books:authorName> $authorName
and $essay <rdf:type> $type
and (trans($type <rdfs:subClassOf> <books:Essay>)
or $type <tks:is> <books:Essay>)
```

As illustrated in the query, iTQL provides the function `trans` as means for computing transitive closure of a relation (such as `rdfs:subClassOf`) and therefore also resources not directly classified as `books:Essay` but rather as one of its subclasses are returned. Paths of arbitrary length in the graph can be traversed using another special function called `walk`. The above query could also be expressed using `walk`.

Worth mentioning is the ability to sort the resulting answers and to provide access to answers in a paged mode using `limit` and `offset` as in SQL.

Also, in contrast to the SquishQL-family of query languages discussed in Section 4.3.1, iTQL allows the specification of nested queries.

Project page:

http://www.kowari.org

Implementation(s):

production use implementation as part of the Kowari Metastore and used in the commercial product Tucana Knowledge Server

Online demonstration:

none

4.3.5.3 N3QL

A restricted subset of Notation 3 [49] (short N3), an alternative syntax and extension for RDF that introduces rules, variables, and quoting for easy expression of statements about statements.

Although, as noted in [48], the rules mechanism provided in N3 allows for similar capabilities as those expected from a query language, [48] proposes a syntax for a query language using more conventional means such as `select-where` clauses.

The essential difference between N3QL and most of the other query languages for RDF discussed in this survey, is that a query is an N3 expression and all “keywords” in the query are actually RDF properties of an RDF node representing the query (usually a blank node, but it is also allowed to assign identifiers to queries). To illustrate this difference, consider Query 1 and its realization in N3QL:

```
@prefix books: <http://example.org/books#>.
@prefix n3ql: <http://www.w3.org/2004/ql#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
[] n3ql:select { n3ql:result n3ql:is (?book ?author ?authorName) };
n3ql:where { ?book rdf:type books:Essay;
?book books:author ?author;
?author books:authorName ?authorName }.
```

The result of such a query is the RDF graph specified in the `n3ql:select` clause, in this case a set of RDF collections (indicated by the collection constructor `()`) containing a binding for each of the three selected variables.

[48] seems to indicate that the semantics of such a query is equivalent to the semantics of a rule where the `where` part of the query is the premise of the rule and the `select` part the implication. However, N3 rules can be used to implement, e.g., the transitive closure of an RDF property:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
{?x rdfs:subClassOf ?z; ?z rdfs:subClassOf ?y}
=> {?x rdfs:subClassOf ?y}
```

Such an ability is not attributed to queries written in the above syntax.

Since the description of N3QL is very sketchy, it is hard to evaluate its expressive power: In particular, it is not clear which of the syntactic constructs of N3 can be used in N3QL. [48] states that N3QL is a restricted form of N3 where formulae cannot be nested and literals cannot be subjects of statements.

In particular, the N3 syntax for anonymous nodes, for navigating in the RDF graph using path expressions, and for quantifying variables is assumed to be available in N3QL too. This allows for very concise formulation of queries such as “Return all books written by an author with name ‘Julius Caesar’.”:

```
@prefix books: <http://example.org/books#>.
@prefix n3ql: <http://www.w3.org/2004/ql#>.
[] n3ql:select { n3ql:result n3ql:is (?book) };
n3ql:where { ?book!books:author!books:authorName ‘‘Julius Caesar’’ }.
```

Project page:
`http://www.w3.org/DesignIssues/N3QL.html`

Implementation(s):
prototype, CWM `http://www.w3.org/2000/10/swap/doc/cwm.html`, EulerSharp `http://eulersharp.sourceforge.net/2003/03swap/`

Online demonstration:
none

4.3.5.4 PerlRDF Query Language

PerlRDF[7] is a collection of APIs for parsing, storing, and querying RDF developed by Ginger Alliance[8]. As part of this project, also an RDF query language has been specified. The details of this language are described in [12].

The query language provides path expressions similar to RQL's general path expression and uses a familiar Select...From...Where syntax. An implementation of Query 1 modified to return only those books with a translator named "J. M. Carter" in this query language is shown in the following:

```
Select ?book, ?author, ?author→books:authorName
From books:Essay::?book→books:author{?author}
Where ?book→books:translator→books:translatorName=>'J. M. Carter'
Use books For [http://example.org/books#]
```

The query has deliberately written to demonstrate different features of the query language, e.g., the ability to use path expression not only in the From-clause (as, e.g., in RQL), but also in the Where- and even in the Select-clause of the query. Furthermore, syntactic short hands for specifying the type of the ?book variable and the constraint on the literal value selected by the path in the Where clause are used. This query is equivalent to the following one, that does not use any of these advanced constructs

```
Select ?book, ?author, ?authorName
From ?book→books:author{?author}→books:authorName{?authorName},
?book→rdf:type{books:Essay},
?book→books:translator→books:translatorName{?translatorName}
Where ?translatorName = 'J. M. Carter'
Use books For [http://example.org/books#]
rdf For [http://www.w3.org/1999/02/22-rdf-syntax-ns#]
```

The most interesting syntactic short hands are the use of books:Essay::?book to type the ?book variable using a syntax reminiscent of the underlying Perl language and the use of the => operator to mark the so-called "target element" of a path and place a restriction on the (literal) value of that element.

Project page:
http://www.gingerall.com/charlie/ga/xml/p_rdf.xml

Implementation(s):
available from the project page

Online demonstration:
http://rdf-demo.gingerall.cz/charlie/rdf/act/rdf_demo.act

4.3.5.5 R-DEVICE Deductive Language

The R-DEVICE system, presented in [32], is a "deductive object-oriented knowledge-base system for querying and reasoning about RDF metadata" [http://lpis.csd.auth.gr/systems/r-device.html]. It is a reimplementation of the X-DEVICE language [31] in the C Language Integrated Production System (CLIPS, see http://www.ghg.net/clips/CLIPS.html) using the CLIPS Object-Oriented Language (COOL).

The mapping from RDF triples to objects is achieved in the following way:

[7] http://www.gingerall.com/charlie/ga/xml/p_rdf.xml

[8] http://www.gingerall.com/

- All resources are represented as objects where the type is determined by the `rdf:type` property of the resource. For resources that are classified in multiple classes a dummy class represents a common subclass of all the classes the resource is classified in.
- Properties are realized as multi-slots (slots with multiple values) in the class that is the domain of the property. If no domain is given the property can be applied to any resources, therefore is added as a slot to the class representing `rdfs:Resource` (the top of the resource object hierarchy).

New assertions (generated, e.g., through rules) can require dynamic class and/or object re-definitions.

To illustrate the syntax of R-DEVICE consider the following implementation of Query 1:

```
(deductiverule q1
?book <- (? (rdf:type books:Essay) (books:author ?author))
?author <- (? (books:authorName ?authorName))
=>
(result (book ?book) (author ?author) (authorName ?authorName))
)
```

Note, the production-rule like syntax of R-DEVICE. R-DEVICE also provides constructs for traversing arbitrary length paths of slots and objects (properties and resources) both with and without restriction on the type of slot that may be traversed. This allows to implement both Query 2, where we want to collect all things related to the book with title "Bellum Civile" (this is indicated in R-DEVICE by an unconstrained path of arbitrary length from the book `?book` to the related resources `?related`),

```
(deductive rule q2
?book <- (? (rdf:type books:Essay) (books:title ‘‘Bellum Civile’’) (($?p) ?related)
=>
(result (book ?book) (related ?related))
)
```

and Query 8, where the transitive closure of the `rdfs:subClassOf` relation is to be computed. The latter query can be expressed using a recursive sub-path `rdfs:subClassOf`.

Project page:
`http://lpis.csd.auth.gr/systems/r-device.html`

Implementation(s):
available from project page

Online demonstration:
none

4.3.5.6 RDF-QBE

In [320] a language for querying RDF graphs following the well-established "query by example" paradigm [378, 379] is proposed. Essentially, an RDF graph (described in Notation 3 syntax [49]) is used to describe the query pattern that should be found in the data. Variables in the pattern are expressed as blank nodes without explicit node identifiers as described in [234]. This leads to a major restriction of the approach: query patterns may only form a tree not a graph.[9]

Query 1 can be expressed in RDF-QBE as

```
[] a books:Essay; books:author [ books:authorName [] ].
```

[9] Contrary to the claim in [320], this does however not reduce the problem to tree matching, as the data is still graph shaped.

When considering what this query should return, the problem of handling blank (or anonymous) nodes in RDF must be addressed. Either some identifier (scoped only within the collection of data considered at the moment, for more details see [234]) is assigned to the blank node or the blank node is represented as the collection of its properties. This issue actually has to be considered by all RDF query languages, in particular as assigning an identifier to blank nodes does not mean that blank nodes can be treated as a node with explicit identifier.

In general, RDF-QBE provides a very convenient, easy to read syntax, but the trade-off (acknowledged in [320]) is the rather low expressiveness of the language, as detailed in Appendix B.

Project page:
none

Implementation(s):
described in [320], but not publicly available

Online demonstration:
none

4.3.5.7 RDFQL

RDF Gateway [214] is a platform for developing and deploying Semantic Web applications combining a "native" RDF database engine with a Web server and a server-side scripting language. The RDF database engine allows for the integration of standard and Semantic Web using so-called "virtual tables" and inference rules for deductive reasoning (so far, libraries for OWL and RDFS are provided). A graphical editor for RDF graphs statements is provided for easy creation of new data. To enable basic interoperability with different Semantic Web tools, several RDF serialization formats are supported, viz. RDF/XML, N3 and NTriples (cf. Appendix A).

The RDF Gateway uses a proprietary query language, referred to as RDFQL, described in [2]. Although in many ways similar to RDQL there are several noteworthy differences:

- Transaction management is realized in RDFQL by database commands for starting and committing or undoing (rollback) of a transaction.

- SQL-like update commands are provided, including a full data definition language.

- Data can be stored in data sources (often referred to as "tables", although they differ from tables in SQL database by having a fixed schema as they are only meant to store RDF triples or quads) that can be either disk-based, in-memory or external data sources identified, e.g., by an URI.

- Using the command `INFER`, deductive rules can be defined as part of a `RULEBASE` to be used when querying. This allows, e.g., to specify the semantics of RDFS in the following way (note, that an RDF statement with subject *S*, predicate *P*, and object *O* is written in RDFQL as `{?P ?S ?O}`, i.e., in prefix notation; note also the use of `uri(?u)=?u` to detect whether the object of an RDF statement is a resource (in which case it has an URI and that URI is equal to the "value" of the resource itself) or a literal):

```
RULEBASE rdfs
{
INFER {[rdf:type] ?a [rdf:Property]} from {?a ?x ?y};
INFER {[rdf:type] ?x ?z} from {[rdfs:domain] ?a ?z} and {?a ?x ?y};
INFER {[rdf:type] ?u ?z} from {[rdfs:range] ?a ?z}
and {?a ?x ?u} and uri(?u)=?u;
INFER {[rdf:type] ?x [rdfs:Resource]} from {?a ?x ?y};
INFER {[rdf:type] ?u [rdfs:Resource]} from {?a ?x ?u} and uri(?u)=?u
INFER {[rdfs:subPropertyOf] ?a ?c}
from {[rdfs:subPropertyOf] ?a ?b} and {[rdfs:subPropertyOf] ?b ?c}
INFER {?b ?x ?y} from {[rdfs:subPropertyOf] ?a ?b}
and {?a ?x ?y}
INFER {[rdfs:subClassOf] ?x [rdfs:Resource]}
from {[rdf:type] ?x [rdfs:Class]}
INFER {[rdfs:subClassOf] ?x ?z} from {[rdfs:subClassOf] ?x ?y}
and {[rdfs:subClassOf] ?y ?z}
INFER {[rdf:type] ?a ?y} from {[rdfs:subClassOf] ?x ?y}
and {[rdf:type] ?a ?x}
}
```

Query 1 can be implemented in RDFQL by the following program:

```
session.namespaces["books"] = "http://example.org/books#";
var booksdata = new DataSource("http://example.org/books");
SELECT ?essay, ?author, ?authorName USING booksdata WHERE
{[rdf:type] ?essay [books:Essay]}
and {[books:author] ?essay ?author}
and {[books:authorName] ?author ?authorName}
ORDER BY ?authorName DESC;
```

Note again the (in the RDF context) uncommon notation of statements with predicate first. For illustration of the ability of RDFQL to return ordered result, an `ORDER BY` clause is added to the query that orders the result by the name of the author. Using a rule base as the one shown above for RDFS, even resources classified by a sub-class of `books:Essay` will be returned.

Project page:
`http://www.intellidimension.com/`

Implementation(s):
within the RDF Gateway, personal edition (limited to non-commercial use and the number of connections allowed) available from project page

Online demonstration:
none (however, the project page can serve as a show case as it is implemented using RDF Gateway)

4.3.5.8 TRIPLE

TRIPLE [341, 342, 203] is a rule-based query, inference, and transformation language for RDF data based upon ideas published in [138] and with a syntax close to F-Logic [230]. The use of F-Logic for querying semi-structured data such as XML or RDF is natural, as one of the strengths of F-Logic approaches is the ability to handle data without a fixed schema demonstrated, e.g., in [250]. Other F-Logic based approaches are, e.g., XPathLog (see Section 4.2.1.7) and the commercial ontology management platform Ontobroker[10].

TRIPLE has been designed to address in particular two weaknesses of previous approaches for RDF query languages:

- Most previous proposals provide a number of predefined constructs implementing the specific semantics of, e.g., RDFS or OWL. The disadvantage of such an approach is the lack of extensibility, although extensibility is a crucial feature of the underlying representation formalism RDF. In contrast, TRIPLE only offers a basic, rule-based language for Horn logic, that is in large parts

[10] `http://www.ontoprise.de/products/ontobroker`

identical to F-Logic [230]. This language can be used, where possible, to implement the semantics of, e.g., RDFS. Where Horn logic is not sufficient, as, e.g., in the case of OWL, TRIPLE is designed to be extended by external modules implementing, e.g., an OWL reasoner.

Building upon [238, 239] for expressing Topic Maps in RDF and [275] for representing UML in RDF, the authors argue in [342] that TRIPLE could also be used to query other representation formalisms for metadata other than RDF.

However, this claim is not demonstrated and, although recent work [178] gives some impression of an integrated query language for RDF and Topic Maps, the adequacy of TRIPLE for querying Topic Maps is questionable in light of the rather awkward mappings from Topic Maps to RDF proposed so far (e.g., many of these approaches result almost exclusively in reified statements).

- Due to the foundation in Horn logic, TRIPLE provides not only a well-defined semantics but also fairly powerful reasoning capabilities (that can be further enhanced by extension modules), both in contrast to previous approaches. In particular, the use of a rule language for both querying and reasoning about the queried data is a natural choice, even more so in the context of the Semantic Web.

Following [342] one can identify a number of areas where TRIPLE differs from basic Horn logic (and logic programming languages such as Prolog). Most of these differences are either related to specificities of the RDF data model or the choice of representing properties similar to slots in F-Logic:

- **Identifying resources:** Resources are identified in RDF by URIs. TRIPLE supports both namespaces and general resources abbreviations (e.g., `isa := rdfs:subClassOf` to simplify the notation of URIs. TRIPLE assumes that all resources are identified properly by an URI, anonymous resources are not considered so far (there is some indication that this will change in the future in [203]).

- **Representing and querying statements:** RDF statements are represented as slots and slot values of the subject in a statement, i.e., `*subject*[*predicate* -> *object*]`. This allows for easy grouping and nesting of statements. As in F-Logic, Path expressions inspired by [169] can be used to traverse several predicates at once.

- **Reification:** In contrast to many other RDF query languages, TRIPLE provides concise support for reified statements by enclosing such statements in angle brackets, e.g., `Julius_Caesar[believes-><Junius_Brutus[friend-of -> Julius_Caesar]>]`.

- **Explicit model specification:** Similar to the module syntax in some F-Logic systems, TRIPLE allows the explicit specification of the model in which a statement or atom is true. The model is again identified by an URI and appended to the statement or atom by `@`.

Finally, one should note that TRIPLE differs from common logic programming languages such as Prolog in requiring all variable to be explicitly quantified.

With this syntax RDF statements and queries can be expressed in TRIPLE. Assuming the data from Figure 2.1 has been loaded as part of a model identified by `http://example.org/books` the following TRIPLE program implements Query 1:

```
rdf := ’http://www.w3.org/1999/02/22-rdf-syntax-ns#’.
books := ’http://example.org/books#’.
booksModel := ’http://example.org/books’.
FORALL B, A, AN result(B, A, AN) <-
B[rdf:type →books:Essay;
books:author →A[books:authorName →AN]]@booksModel.
```

Note, the use of both nesting and grouping of statements for this query. In this formulation, this query selects only resources directly classified as `books:Essay`, below it is discussed how this query can be modified to properly select all resources classified as `books:Essay` or any of its sub-classes in the RDFS subsumption hierarchy.

As discussed above, the specific semantics of different RDF vocabularies such as RDFS or OWL is provided on top of the basic language layer, either as external modules or implemented using the Horn logic reasoning provided by TRIPLE. In [342] the following implementation of RDFS in TRIPLE is given:

```
rdf := ’http://www.w3.org/1999/02/22-rdf-syntax-ns#’.
rdfs := ’http://www.w3.org/2000/01/rdf-schema#’.
type := rdf:type.
subPropertyOf := rdfs:subPropertyOf.
subClassOf := rdfs:subClassOf.
FORALL Mdl @rdfschema(Mdl) {
transitive(subPropertyOf).
transitive(subClassOf).
FORALL O,P,V O[P→V] <-
O[P→V]@Mdl.
FORALL O,P,V O[P→V] <-
EXISTS S S[subPropertyOf→P] AND O[S→V].
FORALL O,P,V O[P→V] <-
transitive(P) AND EXISTS W (O[P→W] AND W[P→V]).
FORALL O,T O[type→T] <-
EXISTS S (S[subClassOf→T] AND O[type→S]).
}
```

Note, that this implements only part of the RDFS semantics, e.g., inference from range and domain restrictions of properties is not provided. However, this is not due to a a limitation of TRIPLE, as adding the following two rules completes the implementation of RDFS in TRIPLE.

```
FORALL S,T S[type→T] <-
EXISTS P, O (S[P→O] AND P[rdfs:domain→T]).
FORALL O,T O[type→T] <-
EXISTS P, S (S[P→O] AND P[rdfs:range→T]).
```

With these rules, the implementation of Query 1 in TRIPLE shown above only needs to be modified with respect to the model it is evaluated against: instead of `@booksModel`, `@rdfschema(booksModel)` is used, that is the original model “expanded” with the above rules implementing RDFS semantics.

In [342], not only the textual syntax for TRIPLE discussed so far is proposed but also an RDF (and therefore) XML representation of the language itself (more precisely, for the basic language layer).

Easy of use. Although TRIPLE employs a syntax less familiar to the average query programmer than languages inspired by main-stream query languages such as SQL or XPath, the close alignment with F-Logic allows users knowledgeable in logic programming languages to become acquainted with the language quickly. However, the lack of a visual syntax or a natural language interface makes it hard for non-experts to formulate. An RDF (and therefore XML) syntax is provided that can indeed be queried by TRIPLE itself.

As discussed above, TRIPLE allows the explicit specification of the model(s) for which a statement or formula should hold. Even more, reasoning methods beyond the Horn logic formulae provided by the basic TRIPLE language can be implemented by external modules.

The most striking weakness of the TRIPLE language is a direct consequence of the generality claimed in [342]. Although data represented in very different formalisms such as RDF, Topic Maps, or UML can be queried in TRIPLE (e.g., by translating the data to RDF), this leads to rather awkward representations of many language features. Even for RDF there are certain aspects of RDF, viz. containers, collections, and anonymous nodes, that are not considered by TRIPLE and can not easily be added to the language.

Functionality—Query Types. As just discussed, TRIPLE's generality is in some cases paid for by a lack of adequacy for the data representation formalisms claimed to be supported. Regarding the different query classes discussed in Section 3.2, one can observe, that most of them can be expressed in TRIPLE as already demonstrated for Query 1 and 8. However, the language does not support aggregation.

Semantics. In [342] the semantics of TRIPLE is given by a mapping of the TRIPLE-specific features to standard Horn logic expressions.

Complexity and implementation. No formal complexity study of the language has been published. So far, only a single prototype implementation of TRIPLE is available.

Reasoning. As discussed above, TRIPLE not only offers full Horn logic reasoning as part of the basic language but is designed to allow the extensions with specific reasoners, e.g., for handling OWL ontologies.

Ontology awareness. Although the basic language does not provide any ontology-specific abilities, ontology awareness can be obtained either by implementing the semantics of the ontology language in the basic TRIPLE Horn logic (as demonstrated for RDFS above) or by extending the language with specific reasoners tailored for the needs of the ontology language such as OWL.

Project page:
`http://triple.semanticweb.org/`

Implementation(s):
available from the project page

Online demonstration:
`http://ontoagents.stanford.edu:8080/triple/` (not functional at the time of writing), some information about projects realized with TRIPLE demonstrating its abilities are available from the project page

4.3.5.9 WQL

Ivanhoe [243] is a frame-based API following ideas from [212, 240] for the Nokia Wilbur Toolkit [245], a collection of APIs for processing XML, RDF, and DAML written in CLOS (Common Lisp Object System) and introduced in [241]. In Ivanhoe, resources described using RDF and/or DAML are represented as frames with a slot for each property of the resource. The (possibly multiple) values of a slot correspond to objects of RDF statements with the resources represented by the frame as subjects. In [244], a comparison of a subset of the Ivanhoe API, referred to as Wilbur Query Language or WQL, is described along the criteria from [200]: Three variants of WQL are discussed:

- the basic path-based query language that allows the selection of some resource reachable by a path via `value`, the selection of all resources reachable by a path via `all-values`, and the test whether two resources are connected by a certain path using `relatedp`;

- the embedding of that query language into Common Lisp (abbreviated as WQL+CL). It is not clear, if there is a restriction on the allowed Common Lisp expressions for this language layer.

- based upon the “transparent” (or “hidden”) inference extensions for Wilbur presented in [242], the final layer, abbreviated WQL+CL+inference, uses WQL+CL as query language but assumes a data store providing inferencing, e.g., for implementing RDFS.

For this survey we concentrate on the first query language WQL only, as WQL+CL is more akin to a programming language than a query language. However, where appropriate the “transparent inferencing” provided by Wilbur is considered when evaluating WQL.

Consider a query that returns the labels of all classes the book with identified by `http://example.org/books#Bellum_Civile` is classified under on the data from Figure 2.1:

```
(setf *db* (make-instance ’db))
(load-db (make-url "http://example.org/books")
:locator "http://example.org/books")
(add-namespace "books" "http://example.org/books#")
(all-values !"http://example.org/books#Bellum_Civile"
’(:seq !rdf:type (:seq (:rep* !rdfs:subClassOf) !rdfs:label)))
```

Note, the use of the `:seq` operator for constructing a sequence of slots (or relations in RDF terms) to be traversed by the query and of the `:rep*` operator for traversing the transitive closure of a slot/relation. `all-values` returns all resources (represented as frames) reachable by the specified path from the source frame, i.e., the frame identified by `http://example.org/books#Bellum_Civile`.

Project page:
Wilbur Toolkit: `http://wilbur-rdf.sourceforge.net/`

Implementation(s):
available from the project page

Online demonstration:
none

4.4 Topic Maps Query Languages

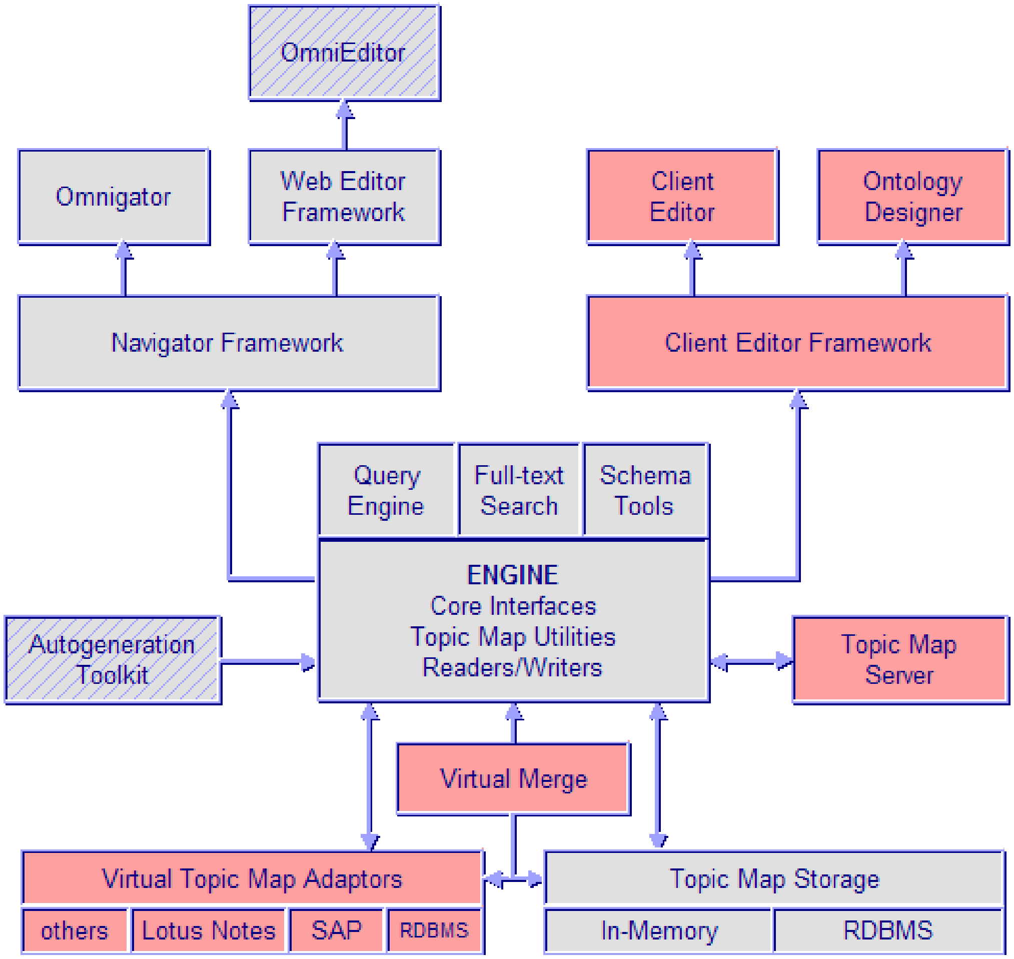
4.4.1 tolog: Logic Programming for Topic Maps

tolog [174] has been developed as part of the Ontopia Knowledge Suite[11], providing access to the core query engine (cf. Figure 4.2). It has also recently (April 2004) been selected as an initial straw-man for the ISO Topic Maps Query Language currently under development. The language is specified in [174] (incomplete), a gentle introduction is presented in the tolog tutorial [175], and in [172, 173] language design and evolution are addressed.

The design is most notably influenced by logic programming languages such as Prolog and logic-based query languages such as Datalog. Some of the more common syntactic constructs have been adapted to an SQL-like feeling to appeal to query authors without background in logic programming. The influence from logic programming is obvious when considering the basic query constructs used in tolog (here and in the following, this discussion of tolog is mostly oriented at [175], since the specification is in many points still unfinished; where possible, extensions described in [173] have been considered):

[11] `http://www.ontopia.net/solutions/products.html`

Figure 4.2 Overview of the Ontopia Knowledge Suite
(from `http://www.ontopia.net/solutions/products.html`, © Ontopia)



- **Identifiers.** tolog provides several means for identifying a Topic Maps construct, most notably based on the (internal) ID of a topic and its subject indicator. E.g. the topic (type) “Novel” in the sample data could be addressed either using its ID, i.e., `Novel`, or its subject indicator `i"http://example.org/books#Novel"`. Note the `i` prefix in the latter case to distinguish between different identifiers used in a topic map. As usual, URI prefixes can be used to abbreviate such expressions, e.g., under the prefix definition `using books for i"http://example.org/books#"` one can write simply `books:Novel` to address that topic by its subject indicator. Note, that these prefixes are *not* equivalent to (XML) namespaces, as also the indicator which of the topic identifiers to use is included.

- **Variables.** In contrast to Datalog or Prolog, variables are prefixed with a `$` wherever they are referenced (this is both for simplicity and to be able to allow uppercase topic IDs). Per default all variables occurring in a query are returned, however using `select $var1, $var2, ...  from` *`query`* a projection on the variables `$var1, $var2, ...` can be obtained.

- **Predicates.** The original tolog proposal (cf. [174]) provides for two kinds of predicates: built-in and dynamic association predicates. For each association type occurring in a topic map, there is a so-called dynamic association predicate that allows querying the extent of the association type. E.g., to query the authors of book `b1` in the sample topic map shown in Chapter 2, one can write `authors-for-book(b1, $AUTHOR: author)`. Note, the use of the association role to identify which of the two associated topic is the author. In this case, the query processor might be able to infer the type of `$AUTHOR` from the type of **b1** and the fact that there are only two topics involved in an `authors-for-book` association. However, in a query such as `authors-for-book($BOOK, $AUTHOR)` the need for specifying the association roles is obvious. Analogously to dynamic *association* predicates there are also dynamic *occurrence* predicates.

  The only two built-in predicates are `instance-of($INSTANCE, $CLASS)` and `direct-instance-of($INSTANCE, $CLASS)`, i.e., predicates that implement the special semantics of the subsumption hierarchy.

  Although the dynamic association and occurrence predicates allow for easy authoring of queries, they require that the query author is aware of the ontology of the data to be queried. This disadvantage has been addressed in a recent proposal on extending tolog [173] and already incorporated into the language tutorial [175]: a number of additional built-in predicates for enumerating the associations, association roles, occurrences, and topics are provided, that allow querying arbitrary topic maps without a-priori knowledge of the types used in the topic maps. E.g., Query 2 can only be implemented using these predicates:

```
select $RELATED from
title($BOOK, "Bellum Civile"),
related($BOOK, $RELATED)?
related($X, $Y) :- {
role-player($R1, $X), association-role($A, $R1),
association-role($A, $R2), role-player($R2, $Y) |
related($X, $Z), related($Z, $Y)
}.
```

  This query also demonstrates the use of conjunctions (denoted by a comma as in prolog), disjunctions (denoted by an expression parenthesized with curly brackets in which the disjuncts are separated by `|`), and inference rules discussed below in detail. Note, the use of the built-in predicates `role-player` and `association-role` that related association roles with topics, respectively associations with association rules.

  Using the notation for disjunctions, one can also specify optional queries.

- **Inference.** As tolog is based on Datalog, it uses a similar syntax for inference rules. E.g., the build-in predicates `instance-of` and `direct-instance-of` can indeed be implemented using only dynamic association predicates and inference rules as follows (cf. [174]):

  ```
  direct-instance-of($INSTANCE, $CLASS) :-
  i"http://psi.topicmaps.org/sam/1.0/#type−instance"(
  $INSTANCE : i"http://psi.topicmaps.org/sam/1.0/#instance",
  $CLASS : i"http://psi.topicmaps.org/sam/1.0/#class").
  super-sub($SUB, $SUPER) :-
  i"http://www.topicmaps.org/xtm/1.0/core.xtm#superclass−subclass"(
  $SUB : i"http://www.topicmaps.org/xtm/1.0/core.xtm#subclass",
  $SUPER : i"http://www.topicmaps.org/xtm/1.0/core.xtm#superclass").
  descendant-of($DESC, $ANC) :- {
  super-sub($DESC, $ANC) |
  super-sub($DESC, $INT), descendant-of($INT, $ANC)
  }.
  instance-of($INSTANCE, $CLASS) :- {
  direct-instance-of($INSTANCE, $CLASS) |
  direct-instance-of($INSTANCE, $DCLASS), descendant-of($DCLASS, $CLASS)
  }.
  ```

  Note, the use of the subject indicator to access the standardized type-instance and superclass-subclass associations.

  Inference rules can also use negation, however according to [173] the semantics of negation in tolog is not yet fully specified.

Aside of these central concepts, tolog also provides constructs for aggregation and sorting ([173] mentions the need for additional aggregation functions), paged queries using `limit` and `offset` clauses similar to SQL, and means for defining and using modules of, e.g., inference rules. Furthermore, recent versions of tolog offer initial support for function libraries on simple data types similar to [256].

Easy of use. The textual syntax of tolog is closely aligned with logic programming languages in the style of Prolog or Datalog. Most of the Topic Maps extensions are rather straight-forward. However, neither a visual syntax nor a natural language interface are provided.

Functionality—Query Types.

Semantics. No formal semantics has been published.

Complexity and implementation. No formal complexity study of the language has been published. Tolog has been implemented in the Ontopia Knowledge Suite[12] and in the open source Topic Maps toolkit TM4J[13]. With tolog selected as initial straw-man for the upcoming ISO Topic Maps Query Language, a more wide-spread adoption of the language is to be expected.

Reasoning. Roughly the same reasoning abilities as in Prolog are provided, although the handling of negation is unclear in the current documents describing tolog.

[12] `http://www.ontopia.net/solutions/products.html`

[13] `http://tm4j.org/`

Ontology awareness. tolog can access and query type information included in a topic map. In particular, the special semantics of the associations defining the subsumption hierarchy is considered. No further support for ontology languages is provided. However, it is likely that the ISO Topic Maps Constraint Language[14] currently under development will be supported in the future.

Project page:

`http://www.ontopia.net/omnigator/docs/query/tutorial.html` (language tutorial)

Implementation(s):

as part of the Ontopia Knowledge Suite (`http://www.ontopia.net/solutions/products.html`) and the open source Topic Maps toolkit TM4J (`http://tm4j.org/`).

Online demonstration:

Omnigator is a show-case application implemented using the Ontopia Knowledge suite. It is available from `http://www.ontopia.net/` and an online demonstration can be accessed at `http://www.ontopia.net/omnigator/models/index.jsp` (however, there seems to be no way to directly test tolog queries in the online demonstrator).

4.4.2 AsTMA?: Functional-style Querying of Topic Maps?

AsTMa? is an "experimental" Topic Maps query language integrated with the rest of the AsTMa language family developed at the Australian Bond University. It is defined in [27]. A language tutorial is also provided [25].

AsTMa? has a rather different flavor compared to other Topic Maps languages as it is most similar to functional XML query languages in the style of XQuery [57]. It specifies several different path languages than can be used for accessing data in topic maps. These data can than be further processed by various query constructs, in particular for constructing XML output.

Query 1 can be implemented by the following AsTMa? query:

```
<books>
{
forall [$book (Writing)] in http://example.org/books
return
<book>
{$book,
forall $author in ($book →author / author-for-book) return
<author>
{$author}
<name>{$author/bn}</name>
</author>
</book>
}
</books>
```

Note the almost identical syntax in comparison to XQuery. The query first selects all topics with topic type `Writing` in `http://example.org/books`. For each of these books, the author is queried by traversing the `author-for-book` association and selecting the topics with `author` role. Finally, the basename of each author is returned by the expression `$author/bn`.

One of the interesting features of AsTMa? is the great variety offered for accessing topics and associations: based on a path expression in one of the path languages or on a constraint written in the AsTMa!. This allows the above query to be formulated as

[14] `http://www.isotopicmaps.org/tmcl/`

```
<books>
{
forall [$book (Writing)] in http://example.org/books
return
<book>
{$book,
forall [ (author-for-book)
Writing : $book
author: $author ]
in http://example.org/books return
<author>
{$author}
<name>{$author/bn}</name>
</author>
</book>
}
</books>
```

Here the association is not queried by a path expression, but rather by a constraint in AsTMa! syntax.

Project page:
`http://astma.it.bond.edu.au/querying.xsp`

Implementation(s):
as part of the Perl XTM module, available via CPAN

Online demonstration:
`http://astma.it.bond.edu.au/query/`

4.4.3 Toma: Querying Topic Maps inspired by SQL

Due to its wide-spread acceptance, designing a query language with syntax and features similar to SQL is a natural choice. A first proposal for a Topic Maps query language inspired by SQL can be found in [237], that allows to query topics, topic-types, and associations using a mixture of SQL syntax and path expressions. E.g., the following query selects all books (i.e., topics classified as `Writing`) together with their authors:

```
select topic[book], topic[author]
from topic-type["Writing"].topic[book],
topic[book]..assoc[a]..topic[author],
assoc-type["author−for−book"].assoc[a]
```

[237] also points out that a close alignment of query language and representation formalism is desirable, e.g., to allow the user to author a query with (nearly) the same tools used for authoring the topic maps itself.

Developed at Space Application Services[15] in a project for the European Space Agency, Toma [308] is a more elaborate proposal for a Topic Maps query language along the lines of SQL and path expressions known from object-oriented query languages. It provides access to all Topic Maps concepts, including an implementation of the special semantics of the subsumption hierarchy. Information about a topic such as topic ID, basename, or subject identifier are accessed using a `.` notation as in object-oriented languages. E.g., `$topic.bn = 'Julius Caesar'` compares the basename of all topics selected by `$topic` with the string "Julius Caesar". Associations can be traversed using `->`, however only the associations with special semantics such as the instance-of or superclass-subclass association can also be traversed transitively, i.e., without knowing the length of the path to be traversed a-priori. Indeed, for traversing the subsumption

[15] `http://www.spaceapplications.com/`

hierarchy Toma provides the notation `$start.super(1..*)` that selects all super-classes of the current one. Instead of `1..*` one can specify any interval or single number to indicate how many superclass-subclass associations shall be traversed. A similar notation is available for instance-of associations.

A Toma expression implementing Query 1 is shown in the following:

```
select $book, $author, $author.bn
where $book.type(1..*).id = 'Writing'
and author-for-book%a→Writing = $book
and author-for-book%a→author = $author
```

This query selects all topics classified as `Writing` or one of its subtypes together with their authors and the basenames of the authors. The link between the book and the author is established by the `author-for-book` associating: a topic *x* is the author of another topic *y*, if *x* occurs in role `author` and *y* in role `Writing` in the same association (the use of `%a` in the query ensures that the same association is used).

Using the above mentioned constructs for querying the type hierarchy, Query 3 can be implemented easily: The following Toma query selects all topics that are neither used as type of another topic (i.e., part of the "ontology") nor typed as `Translator`:

```
select $topic
where $topic.type(1..*).si.sir != 'http://example.org/books#Translator'
and not exists ($t.type(1) = $topic)
and not exists ($t.type(1..*) = $x and $topic.super(1..*) = $x)
```

In this query all topics are selected that neither (a) have the subject identifier `http://example.org/books#Translator` nor (b) are the type of some topic nor (c) are a sub-class of topic that is the type of some topic.

Project page:
`http://www.spaceapplications.com/toma/`

Implementation(s):
implementation not freely available

Online demonstration:
none

4.4.4 Path-based Access to Topic Maps

As for RDF and XML query languages, following the success of XPath great interest in path-based query languages for Topic Maps has been triggered (cf. [26] for an overview of such languages and a plea for inclusion of path navigation in the upcoming ISO Topic Maps query language). Actually, most of the current proposals for Topic Maps query languages use some form of path expressions with the notable exception of tolog (see Section 4.4.1). In the following, two languages focusing on providing navigational path expressions for accessing Topic Maps languages are investigated. Further proposals for path expressions as basis for querying Topic Maps are discussed in Section 4.4.2 and 4.4.3.

4.4.4.1 XTMPath

XTMPath [29] is an approach to define an easy-to-use language for accessing data stored in topic maps using XPath-like path expressions. The language is defined in [29] as part of the Perl XTM toolkit, [199] presents an easy overview. The core principle of the language is to use the way Topic Maps are serialized as XML documents in XTM [306] as orientation on how to address parts of a topic map. E.g., the expression `topic[instanceOf/topicRef/@href = "#Historical_Novel"]` finds all topics that are (directly) typed as a `Historical_Novel`. The path expression results from the way this information is represented in XTM:

```
<topic id="b1">
<instanceOf>
<topicRef xlink:href="#Historical_Novel"/>
</instanceOf>
</topic>
```

So, given an XTM serialization of a topic maps finding the correct paths for addressing a part of that topic map is fairly easy.

However, one has to be aware of certain peculiarities of XTMPath:

- Only a very small subset of the language constructs provided in XPath is currently supported by XTMPath, mostly abbreviated syntax for child and descendant axis and some simple predicates.
- [29] argues that, in contrast to XPath, the XTMPath processor essentially operates on data conforming to a single DTD (viz., the XTM DTD). This observation leads to treating the child axis in most cases as equivalent to the descendant axis. Only in some rare cases (e.g., for `instanceOf`) a difference between child and descendant is made.

Project page:
`http://cpan.uwinnipeg.ca/htdocs/XTM/XTM/Path.html`

Implementation(s):
as part of the XTM toolkit available from CPAN

Online demonstration:
none

4.4.4.2 TMPath

TMPath [59, 60] is an "experimental" language for accessing parts of a topic maps in a style similar to XPath. It has been developed by Dimitry Bogachev, partly as input to the ISO Topic Maps Query Language working group.

Just as XPath, it is designed to be used as embedded language that implements the selection of (or access to) parts of the topic maps, that can be used by the host language for further processing.

The latest version of TMPath [60] provides a large number of constructs for this task. In contrast to the strict syntax of compound steps in XPath consisting in axis and node-test with an optional predicate, TMPath mixes different styles of steps. E.g., the following TMPath expression returns the basename of all authors, i.e., all topics that occur in the `author` role in a `author-for-book` association:

```
/topic[*;roleOf::author[is-author-of]/role::Writing]/bn::*[1]
```

The syntax `*;...` is one of the many shortcuts in TMPath specifying a type condition before the semicolon and arbitrary further conditions after it.

TMPath also provides means for binding variables using `for` clauses familiar from XPath. This allows, e.g., to return a list of all Writings together with their authors:

```
for $book in /topic[subjectIdentifier = "http://example.org/books#Writing"]
for $author in /topic[*;roleOf::author[is-author-of]/role::Writing = $book]
return list{$book/bn::*[1],$author/bn::*[1]}
```

Although obivously inspired by XPath, the lack of strict rules for representing the various TMPath step and predicate expressions leads to a rather complicated and hard to read syntax.

Project page:
`http://homepage.mac.com/dmitryv/TopicMaps/TMPath/TMPathRevisited.html`

Implementation(s):
unclear, not freely available

Online demonstration:
none

4.5 OWL Query Languages

4.5.1 OWL-QL

OWL Query Language (OWL-QL) combines a formal language for querying OWL ontologies with a protocol designed to support a dialogue between a querying agent and an answering agent [163]. It is based on the earlier DAML Query Language (DQL) developed by the Joint US/EU ad hoc Agent Markup Language Committee [162]. Although based on OWL and RDF, the language is quite generic and could easily be adapted to other logic based knowledge representation formalisms such as SWRL [207] or SCL[16].

Query Language In OWL-QL, the query language itself is based on the standard notion of language statements (in this case OWL statements) in which some terms are replaced by variables [126]. Queries will often resemble standard conjunctive queries with the predicates being OWL classes and properties, and constants being OWL individuals [209]. An answer to a query consists of a *binding* for some or all of the variables in the query, i.e., a set of individual names from the ontology(ies) which, when substituted for the corresponding variables, give a set of statements that are entailed by the ontology(ies) being queried. E.g., a query of the form

$$\mathsf{Person}(?x)\ \mathsf{worksFor}(?x, \mathsf{W3C})$$

asks for persons who work for W3C, and if the ontology contains the statements $\mathsf{Person}(\mathsf{Jane})$ and $\mathsf{worksFor}(\mathsf{Jane}, \mathsf{W3C})$, then binding Jane to $?x$ answers the query as under this binding the statements in the query are entailed by the ontology.

As usual, there are different kinds of variable: *must-bind* variables are those which must be be bound to some individual name (in conjunctive queries, these are usually called distinguished variables); *don't-bind* variables are those for which no explicit binding is required (in conjunctive queries, these are usually called non-distinguished variables); and *may-bind* variables are those which may optionally bound to an individual name. The addition of may-bind variables doesn't increase the expressive power of the language (the same result could be achieved by combining the results of queries using only must-bind and don't-bind variables), but may be convenient in some applications. Variables that are not bound to individual names in a query answer are treated as existentially quantified. For full details of the semantics of OWL-QL, the reader is again referred to [163].

In OWL-QL, standard taxonomic queries, e.g., retrieving all the super-classes of a given class, can be answered by using RDF properties in query atoms. For example, the query

$$\mathsf{subClassOf}(\mathsf{Person}, ?x)$$

[16] Common Logic Standard, `http://cl.tamu.edu`.

would return all the super-classes of Person.

Given that the semantics of OWL-QL are based on entailment, answering OWL-QL requires, in general, the use of a theorem prover. One possible technique is to transform both query and ontologies into First Order Logic and use a FOL order theorem prover; details of an implementation based on this idea can be found at the DQL project for the Stanford Knowledge Systems Laboratory (`http://ksl.stanford.edu/projects/dql/`). Another technique is to reduce the conjunctive queries to standard retrieval queries that can be answered using a Description Logic (DL) reasoner [209, 355, 206]; details of an implementation based on this technique can be found at [179]. Details of a Jess based implementation can be found at [339].

Query Answering Protocol The OWL-QL query answering protocol is designed to cope with the wide variety of situations that might arise in a heterogeneous web environment:

- there may be many different kinds of *server* (a query answering service) with access to different kinds of information represented in many different formats;
- servers may have only partial information and may have limitations with respect to their performance (speed and/or completeness), the language they can handle (e.g., OWL Lite/DL/Full) and the kinds of query they are able to answer;
- the querying agent may or may not want or be able to specify all of the ontologies that should be used in answering a query;
- the querying agent may need only one answer, all possible answers, or something in between;
- the querying agent may consider some parts of the answer to be more important than others;
- and querying agents and servers may use different forms of surface syntax, e.g., RDF or XML.

In order to address these requirements, a query answer can be returned in one or more *bundles*, the query agent can specify an upper limit on the size of answer bundles, and a server can indicate the characteristics of its answer with respect to completeness and duplication. The query agent can also specify zero or more ontologies and (optionally) additional OWL statements that are to be used in computing the query answer. Moreover, the query language is specified using an abstract syntax for which many different serialisations are possible (e.g., an XML serialisation has been defined at the DQL project for the Stanford Knowledge Systems Laboratory, `http://ksl.stanford.edu/projects/dql/`).

When the query agent specifies one or more ontology, then only these ontologies can be used to compute answers.[17] Alternatively, the query agent may specify a variable instead of an ontology. In this case the server is free to use any ontology it chooses to answer the query (the idea here is that the server can use arbitrary web accessible resources in order to find answers to such queries); the actual ontology used to answer the query may be returned as a binding for the variable, depending on whether it is a must-bind or may-bind variable.

If the query agent specifies an upper limit on the size of the answer bundle, then the size of an answer bundle returned by the server may range from zero up to this limit. An answer bundle from the server also includes either a process handle or a termination token. In case a process handle is returned, this can be used by a query agent (not necessarily the same agent) to continue the dialogue by requesting more answers to the query; a query agent can also terminate the dialogue at this point by sending the server the process handle with a termination request. A dialogue can be ended by the server using one of three different types of termination tokens: *end* simply indicates that no more answers will be provided; *none*

[17] Note that these ontologies may import others using OWL's import mechanism.

explicitly asserts that all possible answers have been returned (i.e., the union of the answer sets in this and any preceding bundles constitute a complete answer to the query); and "rejected" indicates that the server is unable to answer the query. This last case covers a range of possibilities, including queries being outside the scope of a particular server (e.g., an OWL DL query sent to an OWL Lite server), or simply ill formed.

The specification of OWL-QL envisages servers with different kinds of behaviour regarding the generation of duplicate answers (although it does not specify a language mechanism whereby this information could be communicated to a query agent). A *non-repeating* server is one which guarantees not to duplicate answers during the course of a dialogue; a *terse* server is one that will not return redundant answers, where an answer is considered redundant if it is less specific than another answer (i.e., it has the same bindings for must-bind variables and a subset of the bindings for may-bind variables); a *serially-terse* server is one that will not return answers that are redundant with respect to already returned answers (but answers could be rendered redundant by a subsequent answer).

Project page:
`http://ksl.stanford.edu/projects/owl-ql/`

Implementation(s):
available from the project page

Online demonstration:
`http://onto.stanford.edu:8080/`

Chapter 5

Evaluation Results

In Appendix B, the detailed evaluation of the query languages discussed above is shown. Here, some of the most striking results are highlighted.

The first and most obvious observation that can be derived from the discussion of the query languages in Chapter 4 and the feature evaluation from Appendix B is the great **variety** of proposed Web query languages ranging from path languages providing only the most basic means for data access (XTMPath, Section 4.4.4.1; RDFPath, Section 4.3.2.5) over similarly basic languages for extracting multiple data items at once (RDQL, Section 4.3.1.3; XQL, Section 4.2.1.3), languages with and without ontology support to computationally complete languages with general recursion that are able to query data in any of the representation formalisms considered here (Xcerpt, Section 4.2.2.6; XQuery, Section 4.2.1.5; TRIPLE, Section 4.3.5.8) or constrained natural language for querying Semantic Web data (Metalog, Section 4.3.4.1). This variety is also reflected in the ability of the different query languages to infer data: some languages do not consider inference, most provide at least some restricted form of inference, e.g., means for computing the transitive closure of a relation, some allow for a Datalog-like specification of intensional data.

Also rather obvious from the results, although not commonly acknowledge is that similar approaches occur for XML, RDF, and Topic Maps querying, e.g., basic path-based languages are proposed for all cases and query languages inspired by logic programming languages such as Prolog or Datalog are often the ones providing the riches expressiveness in each area. This is particularly true when considering the reasoning abilities of the surveyed languages: in all areas languages with no actual reasoning at all, with very limited reasoning abilities for implementing specificities of the underlying representation formalism, and with strong reasoning support, e.g., by means of general recursion and Horn logic clauses, can be found.

This points to a common classification scheme for the query languages surveyed so far using the reasoning abilities of a language as distinguishing property. Such a classification scheme is proposed and discussed in the following section.

5.1 A Classification Scheme for Web Query Languages

In the tradition of the seminal papers by Codd [127] and later by Chandra and Harel [111, 112], query languages for (relational) data base have often been characterized by their expressiveness (or completeness under the relational algebra) and evaluation complexity. However, many recent proposals for Web query languages have acknowledge previous results (e.g., in [7]) suggesting that such classification schemes have to be altered for a Web context by providing computationally complete languages (e.g., XQuery [57], XSLT 2.0 [227], Xcerpt [329], XPathLog [270], TRIPLE [342], tolog [173]).

For Edutella [286], an RDF-based peer-to-peer (P2P) infrastructure, a language for exchanging queries, dubbed RDF-QEL, among the nodes in the P2P system has been developed. This language is based on Datalog but to support a wide range of devices and implementations with differing capabilities, five language layers are proposed distinguished by increasing complexity:

- *Rule-less queries* are queries without rules (or equivalent constructs).
- *Conjunctive queries* allow only a single, non-recursive rule per predicate.
- *Disjunctive queries* can use several rules for defining a predicate but may not be recursive in any way.
- *Linear recursive queries* may contain predicates defined by linear recursion.
- *General recursive queries* may contain arbitrary Datalog predicates.

This classification scheme is very useful to estimate the processing capabilities required for evaluating a query. However, considering the results of this survey, as the traditional classifications based on query complexity, it proves not to deliver an interesting and revealing classification of the languages. This can be attributed to the fact that the RDF-QEL classification has been defined for queries not for languages. E.g., the class of linear recursive queries is certainly interesting, however among the languages surveyed here there is no language that supports only linear recursion.

Therefore, a novel classification scheme for Web query languages is proposed here. The core classification feature used in this scheme is the Semantic Web "fitness", the reasoning abilities of a language. Four classes of languages are proposed, the third one divided into two subclasses, yielding the hierarchy depicted in Figure 5.1:

Class 1: Selection-Only Languages. The main characteristic of these languages, a typical example of which is XPath [121], is the lack of construction abilities, i.e., they are only able to specify which part of the input to be *selected* by the query. A direct consequence of this restriction is the missing of means for construction of intermediate results such as views, rules or functions. Also most of these languages operate on a single document (or similar collection of data given by the context).

Although these restrictions limit the expressiveness severely, they also allow for an easy implementation and, at least in some cases, efficient evaluation of the languages. Furthermore, these languages are often used as part of other technologies or even other query languages, hence allowing the use of the same basic access functionality and syntax over a wide range of technologies.

Class 2: Non-recursive Languages. In contrast to the selection-only languages, these class of languages provides some means of *construction*, often realized by nested queries. However, recursion as needed for inference queries and the traversal of arbitrary-length paths in a structure are either missing entirely or only available on some predefined relations (e.g., parent-child relation in XML, but no traversal of arbitrary-length ID/IDREF or XLink relations).

Also, these languages do not specifically support the extended semantics provided by ontology languages such as RDFS or OWL, although in some cases (e.g., RDQL) certain implementations provide very limited ontology support as part of the storage model.

Such languages can express not only selection, extraction, restructuring, and often also reduction queries, but also some inference and combination queries, where a fixed upper bound on the size of the inferred data exists.

Typical examples of this class are, e.g., RDQL [332] and XML-QL [143].

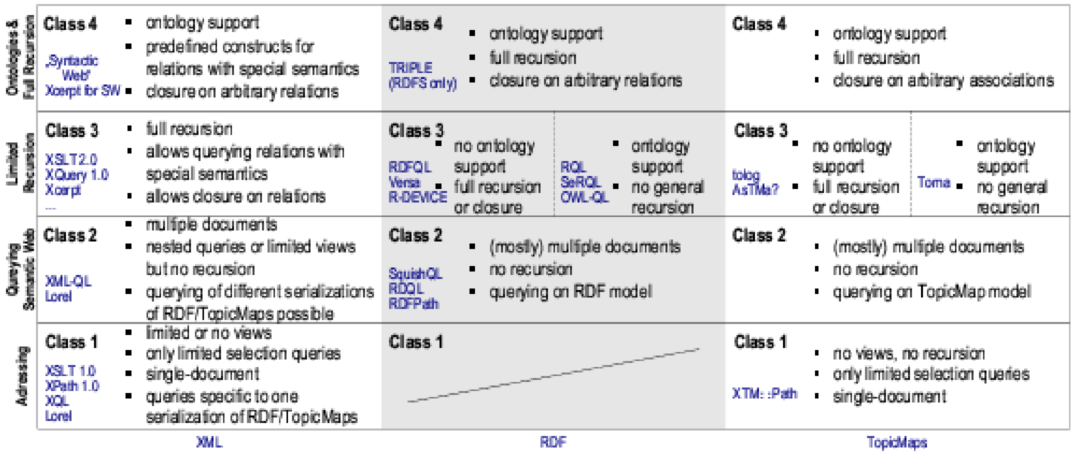
Class 3a: Ontology-aware non-recursive languages. These languages specifically support the use of ontology information for querying, but do not allow the use of general recursion. A well-known example for such a language is RQL [221], which employs ontologies for typing and querying but limits the traversal of arbitrary-length paths to the subsumption hierarchy defined in the ontology.

Class 3b: Recursive languages without specific ontology support. A large number of query languages, in particular for querying XML (e.g., XSLT, XQuery), falls into this class: they provide the ability to express recursive queries on top of the capabilities of Class 2 languages. However, no specific support for ontology reasoning is given. This is not so much a limitation on the expressiveness of the query languages (most of the languages in this class are computationally complete anyway), as on the convenience (and potentially efficiency) for expressing queries relying on ontology information. The mechanisms for inferring knowledge from an ontology describing the queried data have to be explicitly stated in a query.

Often such languages are the basis of extensions (in form of libraries or true language extensions) for supporting ontology reasoning, i.e., the basis of Class 4 languages.

Class 4: Ontology-aware recursive languages. Only languages that support both general recursion (or equivalent operations) and the specificities of some ontology languages such as RDFS or OWL are included in this class. Representative languages are, e.g., Xcerpt, TRIPLE, or tolog.

Figure 5.1 Excerpt of the Sureveyed Languages in Classification Scheme



	XML	RDF	TopicMaps
Ontologies & Full Recursion	Class 4 „Syntactic Web" Xcerpt for SW ▪ ontology support ▪ predefined constructs for relations with special semantics ▪ closure on arbitrary relations	Class 4 TRIPLE (RDFS only) ▪ ontology support ▪ full recursion ▪ closure on arbitrary relations	Class 4 ▪ ontology support ▪ full recursion ▪ closure on arbitrary associations
Limited Recursion	Class 3 XSLT2.0 XQuery 1.0 Xcerpt ... ▪ full recursion ▪ allows querying relations with special semantics ▪ allows closure on relations	Class 3 RDFQL Versa R-DEVICE ▪ no ontology support ▪ full recursion or closure RQL SeRQL OWL-QL ▪ ontology support ▪ no general recursion	Class 3 tolog AsTMa? ▪ no ontology support ▪ full recursion or closure Toma ▪ ontology support ▪ no general recursion
Querying Semantic Web	Class 2 XML-QL Lorel ▪ multiple documents ▪ nested queries or limited views but no recursion ▪ querying of different serializations of RDF/TopicMaps possible	Class 2 SquishQL RDQL RDFPath ▪ (mostly) multiple documents ▪ no recursion ▪ querying on RDF model	Class 2 ▪ (mostly) multiple documents ▪ no recursion ▪ querying on TopicMap model
Adressing	Class 1 XSLT 1.0 XPath 1.0 XQL Lorel ▪ limited or no views ▪ only limited selection queries ▪ single-document ▪ queries specific to one serialization of RDF/TopicMaps	Class 1	Class 1 XTM: :Path ▪ no views, no recursion ▪ only limited selection queries ▪ single-document

This comparison scheme is obviously tailored to querying the kind of data envisioned to be predominant in a Semantic Web setting: Heterogeneous, highly, but often inconsistently structured. Flexible means for programmatic manipulation of such data are called for. This entails, in particular, the ability to query and traverse arbitrary-length paths of related items in the data, both if the relation is expressed in the structure (e.g., parent-child relationship in XML) or established by other means (e.g., ID/IDREF, XLink, based on foreign keys such as URIs).

Traversing arbitrary-length paths of related items is one of the most basic inference query that is required by many use cases proposed for the Semantic vision. Also central to the idea of the Semantic Web is the use of formally defined vocabularies that allow a more precise automated "understanding" of the data described.

The classification scheme proposed in this section uses these two observations for providing an structured view on the languages considered in this survey.

5.2 Observations on the State of the Art of Web Query Languages

Aside of the classification proposed in the previous section, there are a few additional observations on the status of Web query languages that can be derived from the above comparison:

- **Varying Maturity Level:** The query languages surveyed in this report vary noticeably in the level of maturity. As to be expected, query languages for RDF and Topic Maps are in general less evolved than XML query languages that have been investigated in academia and industry for several years. But also, e.g., the proposed RDF query languages differ quite noticeably in their maturity level, some still barely more than quickly drafted proposals, some already in production use.

- **Intense and Early Standardization Activity:** In all three areas, but particularly for XML and Topic Maps query languages standardization activity from various organizations such as W3C or ISO precedes or runs parallel to early implementations and research activity. This can lead to a premature focus on alignment with use cases and requirements proposed by the standardization bodies.

- **Layering of Query Languages:** A possible explanation in the variety observed in the query languages proposals surveyed here is that some of the query languages are limited to a specific task such as the selection of data in a XML, RDF, or Topic Maps structure. Often such limited query languages are than used as a separate access layer in full transformation and query languages. The typical example for such layering is the use of XPath for accessing nodes in an XML document in XSLT, which provides more advanced restructuring and transformation capabilities.

- **Approximation and Reasoning as an Emerging Issue:** At least two issues are receiving increasing interest recently, both motivated by the characteristics of data observed or expected in the (Semantic) Web:

  (1) Not all data is structured properly, quite a lot of interesting information can only be deduced from full-text processing, and there is (not yet) a common understanding of how to structure data properly. This leads to the desire for features in the style of information retrieval systems that (a) allow the processing of full-text data included in the structure and (b) can be extended to allow queries where the structure of the data is only approximately known (e.g., whether a data item is represented as element or attribute).

  (2) Combining such issues with formal vocabularies (from simple thesauri to ontologies described, e.g., in OWL full) requires some ability to reason about the provided data, e.g., for discovering relations between data items not explicitly represented in the structure.

Finally, one should note the similarity of common issues found to be interesting for query languages from all three areas. E.g., in all cases, one has to consider traversing arbitrary-length paths in the relations provided by the representation formalism. Also in all cases, ontology information and similar reasoning techniques can increase the recall of a query in face of heterogeneous descriptions of the data.

Chapter 6

Conclusion and Outlook

This survey presents a unique look on query languages based upon the diverse formats for data representation already used or expected to be used in the (Semantic) Web. It is illustrated that there is a large number of issues common to query languages in each of these areas that goes well beyond general design considerations of query languages but is routed in the characteristics of the Web context. The most prominent issues are the ability to handle heterogeneous data both in structure and content, to support description of the same or similar information types using different vocabularies, to allow incomplete or approximate specifications of queries, and the consideration of reasoning abilities to be able to integrate, mediate, and enrich the data provided.

The results of the evaluation show that a unified classification scheme for XML, RDF, and Topic Maps can be derived that is both meaningful and interesting for understanding the different proposals and their intended usage scenarios. Such a classification scheme together with the detailed results presented in this report can help identifying interesting languages for varying requirements and provides a better insight in the state of the art of Web query languages.

In the perception of the authors, these results stress the need for a query language that is able to handle all these representation formalisms and the plethora of serialization formats proposed for them.

To understand further the requirements for such a language, the REWERSE I4 working group is investigating design principles (a preliminary report can be found in [83]) and use cases for such a query language. Xcerpt [329] represents a first proposal for a language targeted at the flexibility and reasoning capabilities required in this setting.

Acknowledgements. We would like to thank Wolfgang May for reviewing a draft of this deliverable and providing numerous invaluable comments on how to improve both its presentation and content.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Chapter 7

Web and Semantic Web Query Languages

A Survey for the REWERSE Reasoning Web 2005 Summer School

> For the REWERSE Reasoning Web 2005 summer school, a thorough revision of I4-D1 has been used as the basis for a course on Web and Semantic Web query languages. The focus of the revision is on a more concise presentation and on more consistent treatment of the languages. Experimental parts of I4-D1 such as the classification scheme have been omitted.

A number of techniques have been developed to facilitate powerful data retrieval on the Web and Semantic Web. Three categories of Web query languages can be distinguished, according to the format of the data they can retrieve: XML, RDF and Topic Maps. This article introduces the spectrum of languages falling into these categories and summarises their salient aspects. The languages are introduced using common sample data and query types. Key aspects of the query languages considered are stressed in a conclusion.

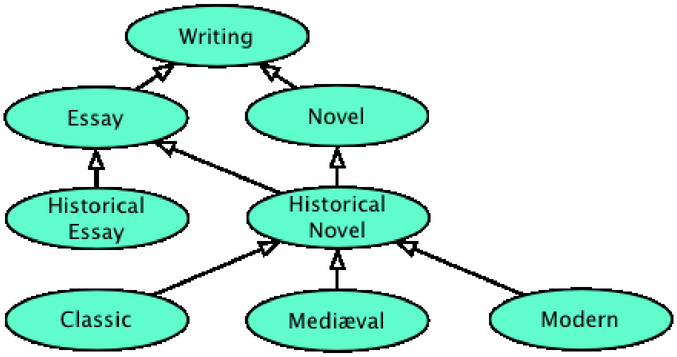
7.1 Introduction

The Semantic Web Vision

A major endeavour in current Web research is the so-called *Semantic Web*, a term coined by W3C founder Tim Berners-Lee in a Scientific American article describing the future of the Web [52]. The Semantic Web aims at enriching Web data (that is usually represented in (X)HTML or other XML formats) by meta-data and (meta-)data processing specifying the "meaning" of such data and allowing Web based systems to take advantage of "intelligent" reasoning capabilities. To quote [52]:

> "The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users."

Figure 7.1 A categorisation of books as it might occur in a Semantic Web ontology



The Semantic Web meta-data added to today's Web can be seen as advanced semantic indices, making the Web into something rather like an encyclopedia. A considerable advantage over conventional encyclopedias printed on paper, however, is that the relationships expressed by Semantic Web meta-data can be followed by computers, very much like hyperlinks can be followed by human readers and programs. Thus, these relationships are well-suited for use in drawing conclusions automatically:

> "For the Semantic Web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning." [52]

A number of formalisms have been proposed for representing Semantic Web meta-data, in particular RDF [288], Topic Maps [215], and OWL (formerly known as DAML+OIL) [35, 210]. These formalisms usually allow one to describe relationships between data items, such as concept hierarchies and relations between concepts. For example, a Semantic Web application for a book store could assign categories to books as shown in Figure 7.1. A customer interested in novels might also get offers for books that are in the subcategory *Historical Novels* and in the sub-subcategories *Classic*, *Mediæval* and *Modern*, although these books are not directly contained in the category *Novels*, because the data processing system has access to the ontology and can thus infer the fact that a book in the category *Mediæval* is also a *Novel*.

Whereas RDF and Topic Maps merely provide a syntax for representing assertions like *"Book A is authored by person B"*, schema or ontology languages such as RDFS [74] and OWL allow one to state properties of the terms used in such assertions, e.g. *"no 'person' can be a 'text'"*. Building upon descriptions of resources and their schemas (as detailed in the architectural road map for the Semantic Web [51]), rules expressed in formalisms like SWRL [208] or RuleML [61] additionally allow one to specify actions to take, knowledge to derive, or constraints to enforce.

Importance of Query Languages for the Web and Semantic Web

The enabling requirement for the Semantic Web is an integrated access to the data on the Web that is represented in any of the above-mentioned formalisms or in formalisms of the "standard Web", such as (X)HTML, SVG, or any XML application. This data access is the objective of Web and Semantic Web *query languages*. A wide range of query languages for the Semantic Web exist, ranging from pure "selection languages" with only limited expressivity, to full-fledged reasoning languages capable of expressing complicated programs, and from query languages restricted to a certain data representation format (e.g.

XML or RDF), to general purpose languages supporting several different data representation formats and allowing one to query data on both the standard Web and the Semantic Web at once.

Structure and Goals of this Survey

This survey aims at introducing the query languages proposed for the major representation formalisms of the standard and Semantic Web: XML, RDF, and Topic Maps. The intended audience are students and researchers interested in obtaining a greater understanding of the relatively new area of Semantic Web querying, as well as researchers already working in the field that want a survey of the state of the art in existing query languages. This survey does *not* aim to be a comprehensive tutorial for each of the approximately 50 languages discussed. Instead, it tries to highlight important or noteworthy aspects, only going in depth for some of the more widespread languages. The following three questions are at the heart of this survey:

1. what are the core *data retrieval capabilities* of each query language,
2. to what extent, and what forms of *reasoning* do they offer, and
3. how are they realised?

7.1.0.1 Structure.

After briefly discussing the three different representation formats XML, RDF, and Topic Maps in Section 7.2.1, each of the languages is introduced with sample queries against a common Semantic Web scenario (cf. Section 7.2.2). The discussion is divided into three main parts, corresponding to the three different data representation formats XML, RDF, and Topic Maps. The survey concludes with a short summary of language features desirable for Semantic Web query languages. The outline is as follows:

1. Introduction
2. Preliminaries
   - 2.1 Three Data Formats: XML, RDF and Topic Maps
   - 2.2 Sample Data: Classification-based Book Recommender
   - 2.3 Sample Queries
3. XML Query and Transformation Languages
   - 3.1 W3C's Query Languages:The Navigational Approach
   - 3.2 Research Prototypes: The Positional Approach to XML Querying
4. RDF Query Languages
   - 4.1 The SPARQL Family
   - 4.2 The RQL Family
   - 4.3 Query Languages inspired from XPath, XSLT or XQuery
   - 4.4 Metalog: Querying in Controlled English
   - 4.5 Query Languages with Reactive Rules

4.6 Deductive Query Languages

4.7 Other RDF Query Languages

5. Topic Maps Query Languages

   5.1 tolog: Logic Programming for Topic Maps

   5.2 AsTMA?: Functional Style Querying of Topic Maps

   5.3 Toma: Querying Topic Maps inspired from SQL

   5.4 Path-based Access to Topic Maps

6. Conclusion

Selection of Query Languages.

This survey focuses on introducing and comparing languages designed primarily for providing efficient and effective access to data on the Web and Semantic Web. In particular, it *excludes* the following types of languages:

- *Programming language tools for XML.* General-purpose programming languages supporting XML as native data type are not considered, e.g. XMLambda [273], CDuce [40], XDuce [211], Xtatic (`http://www.cis.upenn.edu/~bcpierce/xtatic/`), Scriptol (`http://www.scriptol.com/`), and Cω (`http://research.microsoft.com/Comega/` [272]). XML APIs are not considered, e.g.: DOM [18], SAX (`http://www.saxproject.org/`), and XmlPull (`http://www.xmlpull.org/`). XML-related language extensions are not considered, e.g.: HaXML [365] for Haskell, XMerL [371] for Erlang, CLP(Flex) [128] for Prolog, or XJ [201] for Java. General-purpose programming languages with Web service support are also not considered, e.g.: XL [165, 166], Scala [289], Water [309].

- *Reactive languages.* A reactive language allows specification of updates and logic describing how to react when events occur. Several proposals have been made for adapting approaches such as ECA (Event-Condition-Action) rules to the Web, cf. [11] for a survey. There is, of course, a close relationship between such reactive languages and query languages, with the latter often being embedded within the former.

- *Rule languages.* Transformations, queries, consequences, and reactive behaviours can be conveniently expressed using rules. The serialisation of rules for their exchange on the Web is investigated in the RuleML [61] initiative. Similar to reactive languages, rule languages are also closely related to query languages.

- *OWL query languages.* Query languages designed for OWL, e.g., OWL-QL [163], are not considered for two reasons: (1) They are still in their infancy, and their small number makes interesting comparisons hardly possible, (2) the languages proposed so far can only query schemas, i.e., meta-data but not data, and access data only through meta-data, e.g., returning the instances of a class.

A pragmatic approach has been adopted in this survey: A language of one of the above-mentioned four kinds is considered if querying is one of its core aspects, or if it offers a unique form of querying not covered by any of the other query languages considered in the survey. Authoring tools, such as visual editors, are only considered with a query language that they are based upon. The storing or indexing of Web data is not covered (for a survey on storage systems for XML cf. [369], for RDF cf. to [253]).

Despite these restrictions, the number of languages is still quite large. This reflects a considerable and growing interest in Web and particularly Semantic Web query languages. Indeed, standardisation bodies have recently started the process of standardisation of query languages for RDF and Topic Maps. It is our hope that this survey will help to give an overview of the current state of the art in these areas.

7.2 Preliminaries

7.2.1 Three Data Formats: XML, RDF and Topic Maps

7.2.1.1 XML.

Originally designed as a replacement for the language SGML as a format for representing (structured) text documents, XML nowadays is also widely used as a format for representing and exchanging arbitrary (structured) data:

> The "Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML [. . .]. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere."[1]

An "XML document" is a file, or collection of files, that adheres to the general syntax specified in the XML Recommendation [71], independent of the concrete application. XML documents consist of an optional document prologue and a document tree containing elements, character data and attributes, with a distinguished root element.

Elements. Elements are used to "mark up" the document. They are identified by a label (called tag name) and specified by opening and closing tags that enclose the element content. Opening tags are of the form `<label ...>` and contain the label and optionally a set of attributes (see below). Closing tags are of the form `</label>` and contain only the label.

Elements may contain either other elements, character data, or both (mixed content). In analogy with the document tree, such content is often referred to as *children* of an element. Interleaving of the opening and closing tags of different elements (e.g. `<b><i>Text</b></i>`) is forbidden. The order of elements is significant (so-called document order). This is a reasonable requirement for storing text data, but might be too restrictive when storing data items of a database. Applications working with XML data thus often ignore the document order. If an element contains no content, it may be abbreviated as `<label/>`, i.e. the "closing slash" is contained in the start tag.

Attributes. Opening tags of elements may contain a set of key/value pairs called attributes. Attributes are of the form `name = "value"`, where `name` may contain the same characters as element labels and `value` is a character sequence which is always enclosed in quotes. An opening tag may contain attributes in any order, but each attribute name can occur at most once.

References. References of various kinds, (like ID/IDREF attributes and hypertext links) make it possible to refer to an element instead of explicitly including it.

Document Tree. An XML document can be seen as a rooted, unranked[2], and ordered[3] tree, if one does not consider the various referencing or linking mechanisms of XML. Although this interpretation is that

[1] http://www.w3.org/XML/

[2] i.e. the number of children of an element is not bounded.

[3] i.e. the children of an element are ordered.

of the data model retained for XML (cf. XML Infoset [134], XQuery, XPath [159]) and most XML query languages, it is too simplistic. Indeed, references (as expressed, e.g. through ID and IDREF attributes or hypertext links) make it possible to express both oriented and non-oriented cycles in an XML document.

7.2.1.2 RDF and RDFS.

RDF [234, 38] data is sets of "triples" or "statements" of the form (*Subject*, *Property*, *Object*). RDF data is commonly seen as a directed graph, whose nodes correspond to a statement's subject and object and whose arcs correspond to a statement's property (thus relating a subject with an object). For this reason, properties are also often referred to as "predicates". Nodes (i.e. subjects and objects) are labeled by either (1) URIs describing (Web) resources, or (2) literals (i.e. scalar data such as strings or numbers), or (3) are unlabeled, being so-called anonymous or "blank nodes". Blank nodes are commonly used to group or "aggregate" properties. Specific properties are predefined in the RDF and RDFS specifications [258, 234, 204, 74], e.g. `rdf:type` for specifying the type of resources, `rdfs:subClassOf` for specifying class-subclass relationships between subjects/objects, and `rdfs:subPropertyOf` for specifying property-subproperty relationships between properties. Furthermore, RDFS has "meta-classes", e.g. `rdfs:Class`, the class of all classes, and `rdf:Property`, the class of all properties.[4]

RDFS allows one to define so-called "RDF Schemas" or "ontologies", similar to object-oriented data models. The inheritance model of RDFS exhibits some peculiarities: (1) resources can be classified in different classes that are not related in the class hierarchy, (2) the class hierarchy can be cyclic (so that all classes on the cycle are "subclass equivalent"), (3) properties are first-class objects, and (4) in contrast to most object-oriented formalisms, RDF does not describe which properties can be associated with a class, but instead the domain and range of a property. Based on an RDFS schema, "inference rules" can be specified, for instance the transitivity of the class hierarchy, or the type of an untyped resource that has a property associated with a known domain.

RDF can be *serialised* in various formats, the most frequently being XML. Early approaches to RDF serialisation have raised considerable criticism due to their complexity. As a consequence, a surprisingly large number of RDF serialisations have been proposed, cf. [83] for a detailed survey.

OWL [271, 343, 35] extends RDFS with a means for defining description vocabularies for Web resources. OWL is only considered superficially in this survey, cf. Section 7.1.

7.2.1.3 Topic Maps.

Topic Maps [215, 306] have been inspired from work in library sciences and knowledge indexing. The main concepts of Topic Maps are "topics", "associations", and "occurrences". Topics might have "types" that are topics. Types correspond to the classes of object-oriented formalisms, i.e., a topic is related to each of its types in an instance-class relationship. A topic can have one or more "names". Associations are *n*-ary relations (with $n \geq 2$) between topics. Associations might have "role types" and "roles". Occurrences are information resources relevant to a topic. An occurrence might have one or several types characterising the occurrence's relevance to a topic, expressed by "occurrence roles" and "occurrence role types" in the formalism HyTM [215], or only by "occurrence types" in the formalism XTM [306].

"Topic characteristics" denote the names a topic has, what associations it partakes in, and what its occurrences are. "Facets" (a concept of HyTM but not of XTM) are attribute-value pairs that can be attached to any kind of topic map component for explanation purposes. Facets are thus a means to attach to Topic Maps meta-data in another formalism. "Subject identifiers" denote URIs of resources (called "subject indicators" or sometimes also "subject identifiers") that describe in a human-readable form the

[4] this survey tries to use self-explanatory prefixes for namespaces where possible.

subject of a Topic Map component. Commonly, subjects and topics stand in one-to-one relationships, such that they can be referred to interchangeably.

Like RDF data, Topic Maps can be seen as oriented graphs with labeled nodes and edges. Topic Maps offer richer data modeling primitives than RDF. Topic Maps allow relationships, called associations, of every arity, while RDF only allows binary relationships, called properties. Initial efforts towards integrating RDF and Topic Maps are described in [239, 178]. Interestingly, Topic Maps associations are similar to the "extended links" of the XML linking language XLink (`http://www.w3.org/XML/Linking/`).

7.2.2 Running Example: Classification-Based Book Recommender

In the following, we shall consider as a running example queries in a simple book recommender system describing various properties and relationships between books. It consists of a hierarchy (or *ontology*) of the book categories `Writing`, `Novel`, `Essay`, `Historical_Novel`, and `Historical_Essay`, and two books *The First Man in Rome* (a `Historical_Novel` authored by *Colleen McCullough*) and *Bellum Civile* (a `Historical_Essay` authored by *Julius Caesar* and *Aulus Hirtius*, and translated by *J.M. Carter*). Figure 7.2 depicts this data as a (simplified) RDF graph [234, 246, 74]. Note in particular that a `Historical_Novel` is both, a `Novel` and an `Essay`, and that books may optionally have a translator, as is the case for *Bellum Civile*. To illustrate the properties of the different kinds of query languages, the data is in the following represented in the three representation formalisms RDF, Topic Maps, and XML.

The simple ontology in the book recommender system only makes use of the subsumption (or "is-a-kind-of") relation `rdfs:subClassOf` and the instance (or "is-a") relation `rdf:type`. Though small and simple, this ontology is sufficient to illustrate the most important aspects of ontology querying. In particular, querying this ontology with query languages for the standard Web already requires one to model and query this data in an *ad hoc* fashion, i.e. there is no unified way to represent this data. A possible representation is shown in the XML example below.

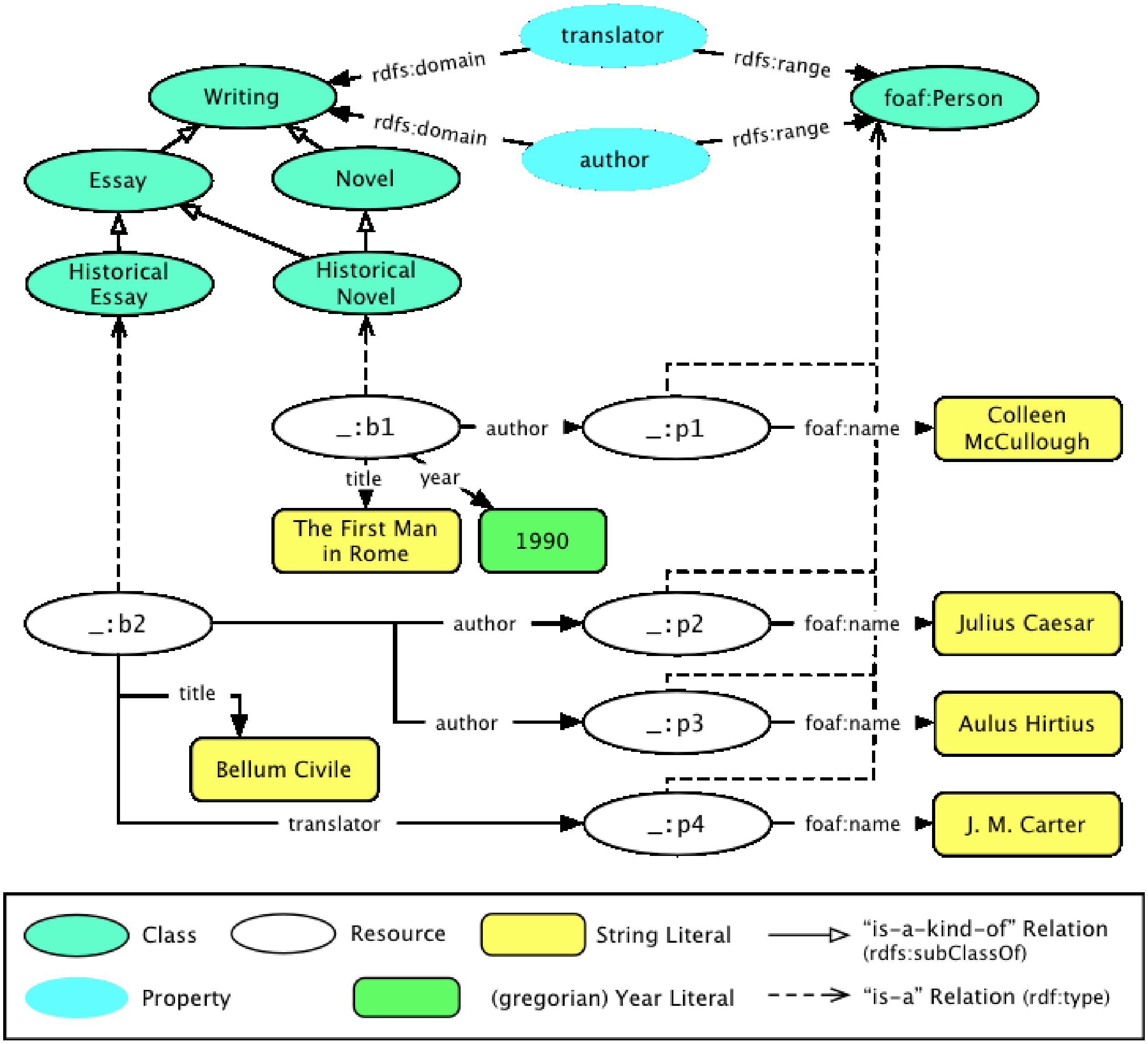
The RDF, Topic Maps, and XML representations of the sample data refer to the "simple datatypes" of XML Schema [54] for scalar data: Book titles and authors' names are "string", (untyped or typed as `xsd:string`), publication years of books are "Gregorian years", `xsd:gYear`. The sample data is assumed to be stored at `http://example.org/books#`, a URL chosen in accordance to RFC 2606 [153] in the use of URLs in sample data. Where useful, e.g when referencing the vocabulary defined in the ontology part of the data, this URL is associated with the prefix `books`.

7.2.2.1 Sample Data in RDF.

The RDF representation of the book recommender system directly corresponds to the simplified RDF graph in Fig. 7.2. It is given here in the *Turtle* serialisation [37].

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .
:Writing a rdfs:Class ;
rdfs:label "Novel" .
:Novel a rdfs:Class ;
rdfs:label "Novel" ;
rdfs:subClassOf :Writing .
:Essay a rdfs:Class ;
rdfs:label "Essay" ;
rdfs:subClassOf :Writing .
:Historical_Essay a rdfs:Class ;
rdfs:label "Historical Essay" ;
rdfs:subClassOf :Essay .
:Historical_Novel a rdfs:Class ;
rdfs:label "Historical Novel" ;
```

Figure 7.2 Sample Data: representation as a (simplified) RDF graph.



```
rdfs:subClassOf :Novel ;
rdfs:subClassOf :Essay .
:author a rdfs:Property ;
rdfs:domain :Writing ;
rdfs:range foaf:Person .
:translator a rdfs:Property ;
rdfs:domain :Writing ;
rdfs:range foaf:Person .
_:b1 a :Historical_Novel ;
:title "The First Man in Rome" ;
:year "1990"^^xsd:gYear ;
:author [foaf:name "Colleen McCullough"] .
_:b1 a :Historical_Essay ;
:title "Bellum Civile" ;
:author [foaf:name "Julius Caesar"] ;
:author [foaf:name "Aulus Hirtius"] ;
:translator [foaf:name "J. M. Carter"] .
```

Books, authors, and translators are represented by blank nodes without identifiers, or with temporary identifiers indicated by the prefix "_:".

7.2.2.2 Sample Data in Topic Maps.

The Topic Map representation of the book recommender system makes use of the Linear Topic Maps syntax [176]. Subclass-superclass associations are identified using the subject identifiers of XTM [306]. For illustration purposes, the title of a book is represented as an occurrence of that book/topic.

```
/* Association and topic types for subclass-superclass hierarchy */
[superclass-subclass = "Superclass−Subclass Association Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#superclass−subclass" ]
[superclass = "Superclass Role Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#superclass" ]
[subclass = "Subclass Role Type"
@ "http://www.topicmaps.org/xtm/1.0/core.xtm#subclass" ]
/* Topic types */
[Writing = "Writing Topic Type" @ "http://example.org/books#Writing" ]
[Novel = "Novel Topic Type" @ "http://example.org/books#Novel" ]
[Essay = "Essay Topic Type" @ "http://example.org/books#Essay" ]
[Historical_Essay = "Historical Essay Topic Type"
@ "http://example.org/books#Historical_Essay" ]
[Historical_Novel = "Historical Novel Topic Type"
@ "http://example.org/books#Historical_Novel" ]
[year = "Topic Type for a Gregorian year following ISO 8601"
@ "http://www.w3.org/2001/XMLSchema#gYear" ]
[Person = "Person Topic Type" @ "http://xmlns.org/foaf/0.1/Person"]
[Author @ "http://example.org/books#author" ]
[Translator @ "http://example.org/books#translator" ]
/* Associations among the topic types */
superclass-subclass(Writing: superclass, Novel: subclass)
superclass-subclass(Writing: superclass, Essay: subclass)
superclass-subclass(Novel: superclass, Historical_Novel: subclass)
superclass-subclass(Essay: superclass, Historical_Essay: subclass)
superclass-subclass(Essay: superclass, Historical_Novel: subclass)
superclass-subclass(Person: superclass, Author: subclass)
superclass-subclass(Person: superclass, Translator: subclass)
/* Occurrence types */
[title = "Occurrence Type for Titles" @ "http://example.org/books#title" ]
/* Association types */
[author-for-book = "Association Type associating authors to books"]
[translator-for-book =
"Association Type associating translators to books"]
[publication-year-for-book =
"Association Type associating translators to books"]
/* Topics, associations, and occurrences */
[p1: Person = "Colleen McCullough"]
[p2: Person = "Julius Caesar"]
[p3: Person = "Aulus Hirtius"]
```

```
[p4: Person = "J. M. Carter"]
[b1: Historical_Essay = "Topic representing the book ’First Man in Rome’"]
author-for-book(b1, p1: author)
publication-year-for-book(b1, y1990)
{b1, title, [[The First Man in Rome]]}
[b2: Historical_Novel = "Topic representing the book ’Bellum Civile’"]
author-for-book(b2, p2: author)
author-for-book(b2, p3: author)
translator-for-book(b2, p4: translator)
{b2, title, [[Bellum Civile]]}
```

The representation given above has been chosen for illustrating query language features. In reality, a different representation might be more natural. For instance, a ternary association connecting a book with its author(s), translator, and year of publication could be used. Also, instead of separate associations for author and translator, use of a generic association between persons and books, and use of roles for differentiation would be reasonable.

7.2.2.3 Sample Data in XML.

XML has no standard way to express relationships other than parent-child. The following is thus one of many conceivable *ad hoc* XML representations of the data in the book recommender system. Its use is obviously highly application-specific.

```
<bookdata xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<book type="Historical_Novel">
<title>The First Man in Rome</title>
<year type="xsd:gYear">1990</year>
<author> <name>Colleen McCullough</name> </author>
</book>
<book type="Historical_Essay">
<title>Bellum Civile</title>
<author> <name>Julius Caesar</name> </author>
<author> <name>Aulus Hirtius</name> </author>
<translator> <name>J. M. Carter</name> </translator>
</book>
<category id="Writing">
<label>Writing</label>
<category id="Novel">
<label>Novel</label>
<category id="Historical_Novel">
<label>Historical Novel</label>
</category>
</category>
<category id="Essay">
<label>Essay</label>
<category id="Historical_Essay">
<label>Historical Essay</label>
</category>
<category idref="Historical_Novel" />
</category>
</category>
</bookdata>
```

For the sake of brevity, the above representation does not express that authors and translators are persons. Note the use of ID/IDREF references for expressing the types (e.g. “Novel”, “Historical_Novel”) of books.

One of the XML-based serialisations of the RDF or Topic Maps representations of the sample data could be used for comparing XML query languages. Instead, in this article, the XML representation given above is used, because these XML-based serialisations of the RDF or Topic Maps representations are awkward, complicated to query, and can yield biased comparisons.

7.2.3 Sample Queries

The different query languages are illustrated using five types of queries against the sample data. This categorisation is inspired by [255] and [124].

7.2.3.1 Selection and Extraction Queries.

Selection Queries simply retrieve parts of the data based on its content, structure, or position. The first query is thus:

Query 20. "Select all *Essays* together with their *authors* (i.e. author items and corresponding names)"

Selection Queries are used in the following to illustrate basic properties of query languages, like the basic means of addressing data, the supported *answer formats*, or the way *related information* (like author names or book titles) is selected and delivered (grouping). *Extraction Queries* extract substructures, and can be considered as a special form of Selection Query. Such queries are commonly found on the Semantic Web. The following query extracts a substructure of the sample data (e.g. as an RDF subgraph):

Query 21. "Select all data items with any relation to the book titled 'Bellum Civile'."

7.2.3.2 Reduction Queries.

Some queries are more concisely expressed by specifying what parts of the data *not* to include in the answer. On the Semantic Web, such *reduction queries* are e.g. useful for combining information from different sources, or for implementing different levels of trust: It might be desirable to create a simple list of books from the data in the recommender system, leaving out ontology information and translators:

Query 22. "Select all data items except ontology information and translators."

7.2.3.3 Restructuring Queries.

In Web applications, it is often desirable to *restructure* data, possibly into different formats/serialisations. For example, the contents of the book recommender system could be restructured to an (X)HTML representation for viewing in a browser, or derived data could be created, like inverting the relation `author`:

Query 23. "Invert the relation *author* (from a book to an author) into a relation *authored* (from an author to a book)."

In particular, RDF requires restructuring for *reification*, i.e. expressing "statements about statements". When reifying, a statement is replaced by three new statements specifying the subject, predicate, and object of the old statement. For example, the statement "*Julius Caesar* is *author* of *Bellum Civile*" is reified by the three statements "the statement has subject *Julius Caesar*", "the statement has predicate *author*", and "the statement has object *Bellum Civile*".

7.2.3.4 Aggregation Queries.

Restructuring the data also includes *aggregating* several data items into one new data item. As Web data usually consists of tree- or graph-structured data that goes beyond flat relations, we distinguish between *value aggregation* working only on the values (like SQL's `max(·)`, `sum(·)`, ...) and *structural aggregation* working also on structural elements (like "how many nodes"). Query 24 uses the `max(·)` value aggregation, while Query 25 uses structural aggregation:

Query 24. "Return the last year in which an author with name 'Julius Caesar' published something."

Query 25. "Return each of the subclasses of 'Writing', together with the average number of authors per publication of that subclass."

Related to aggregation are *grouping* (collecting several data items at some common position, e.g. a list of authors) and *sorting* (extending grouping by specifying in which order to arrange data items). Note that they are not meaningful for all representation formalisms. For instance, sorting in RDF only makes sense for sequence containers, as RDF data in general does not specify order for statements.

7.2.3.5 Combination and Inference Queries.

It is often necessary to *combine* information that is not not explicitly connected, like information from different sources or substructures. Such queries are useful with ontologies that often specify that names declared at different places are synonymous:

Query 26. "Combine the information about the book titled 'The Civil War' and authored by 'Julius Caesar' with the information about the book with identifier `bellum_civile`."

Combination queries are related to inference, because inference refers to combining data, as illustrated by the following example: If the books entitled "Bellum Civile" and "The Civil War" are the same book, and 'if 'Julius Caesar" is an author of "Bellum Civile", then 'Julius Caesar' is also an author of "The Civil War".

Inference queries e.g. compute transitive closures of relations like the RDFS `subClassOf` relation:

Query 27. "Return the transitive closure of the `subClassOf` relation."

Not all inference queries are combination queries, as the following example illustrates:

Query 28. "Return the co-author relation between two persons that stand in author relationships with the same book."

Some query languages have closure operators applicable to any relation, while other query languages have closure operators only for certain, predefined relations, e.g., the RDFS `subClassOf` relation. Some query languages support *general recursion*, making it possible and easy to express the transitive closure of every relation.

7.3 XML Query and Transformation Languages

Most query and transformation languages for XML specify the structure of the XML data to retrieve using one of the following approaches:

- *Navigational approach.* Path-based navigation through the XML data queried.
- *Positional approach.* Query patterns as "examples" of the XML data queried.
- Relational expressions referring to a "flat" representation of the XML data queried.

Languages already standardized, or currently in the process of standardisation by the W3C, are of the first kind, while many research languages are of the second kind. This article does not consider languages of the third kind, e.g., monadic datalog [183, 182] and LGQ [296]. Such languages have been proposed for formalizing query languages and reasoning about XML queries. This article also does not consider special purpose languages like ELog [33] which are not tailored towards querying by humans. Finally, this article does not consider XML query languages focused on information retrieval, e.g., XirQL [171], EquiX [129], ELIXIR [116], XQuery/IR [72], XXL [356], XirCL [279], XRANK [197], PIX [15], XSEarch [130], FleXPath [16], and TeXQuery [14]. Although these languages propose interesting and novel concepts and constructs for combining XML querying with information retrieval methods, they (a) do not easily compare to the other query languages in this survey and (b) mostly do not provide additonal insight on the non-IR features of query languages.

7.3.1 W3C's Query Languages: Navigational Approach

Characteristics of the Navigational Approach. The navigational languages for XML are inspired from path-based query languages designed for relational or object-oriented databases. Most such database query languages (e.g., GEM [375], an extension of QUEL, and OQL [103]) require *fully specified* paths, i.e., paths with explicitly named nodes following only parent-child connections. OQL expresses paths with the "extended dot notation" introduced in GEM [375]: "`SELECT b.translator.name FROM Books b`" selects the name, or component, of the translator of books (note that there must be at most one translator per book for this expression to be legal).

Generalized Path Expressions. Generalized (or regular) path expressions [169,117], allow more powerful constructs than the extended dot notation for specifying paths, e.g., the Kleene closure operator on (sub-)paths . As a consequence and in contrast to the extended dot notation, generalized path expressions do not require explicit naming of all nodes along a path.

Lorel. Lorel [6] is an early proposal for a query language originally designed for semistructured data, a data model that was introduced with the "Object Exchange Model (OEM)" [303,180], and can be seen as a precursor of XML. Lorel's syntax resembles that of SQL and OQL, extending OQL's extended dot notation to generalized path expressions. Lorel provides a means for expressing:

- Optional data: In Lorel, the query `SELECT b.translator.name FROM Books b` returns an empty answer, whereas in OQL it causes a type error, if there is no translator for a book.
- Set-valued attributes: In Lorel, `b.author.name` selects the names of all authors of a book, whereas in OQL it is only valid if there is only a single author.
- Regular path expressions, e.g. a (strict) Kleene closure operator for expressing navigation through recursively defined data structures and alternatives in both labeling and structure.

The following Lorel query expresses Query 20 against the sample data (treating attributes as sub-elements since OEM has no attributes):

```
select xml(results:(
select xml(result:(
select B, B.author
from bookdata.book B
where B.type = bookdata.(category.id)+
) ) ) )
```

Lines 1 and 2 are constructors for wrapping the selected books and their authors into XML elements. Note the use of the strict Kleene closure operator + in line 5. Note also that Lorel allows entire (sub-) paths to be repeated, as do most query languages using generalized path expressions.

To illustrate further aspects of Lorel, assume that one is only interested in books having "Julius Caesar" either as author or translator. Assume also that, as in some representations of the sample data, cf. 7.2.2, the literal giving the name of the author is either wrapped inside a `name` child of the `author` element, or directly included in the `author` element. Selection of such books can be expressed in Lorel by adding the following expression to the query after line 5 `B.(author|translator).name? = "Julius Caesar"`.

StruQL. StruQL [164, 168] relies on path expressions similar to that of Lorel. StruQL is another early (query and) transformation language for semi-structured data using Skolem functions for construction.

7.3.1.1 Data Selection with XPath

XPath is presented in [121] and [354, 340], as well as many online tutorials. It was defined originally as part of XSL, an activity towards defining a stylesheet language for XML (in replacement of SGML's stylesheet language DSSSL). XPath provides expressions for selecting data in terms of a navigation through an XML document. In contrast to the previous approaches based on generalized path expressions, XPath provides a syntax inspired from file systems, aiming at simplicity and conciseness. Conciseness is an important aspect of XPath, since it is meant to be embedded in host languages, such as XSLT or XPointer. Other aspects such as formal semantics, expressiveness, completeness, and complexity, have not played a central role in the development of XPath but have recently been investigated at length.

Data model. An XML document is considered as an ordered and rooted tree with nodes for elements, attributes, character data, namespaces declaration, comments and processing instructions. The root of this tree is a special node which has the node for the XML document element as child. In this tree, element nodes are structured reflecting the element nesting in the XML document. Attribute and namespace declaration nodes are children of the node of the element they are specified with. Nodes for character data, for comments, and for processing instructions are children of the node of the element in which they occur, or of the root node if they are outside the document element. Note that a character node is always "maximal", i.e., it is not allowed that two character data nodes are immediate siblings. This model resembles the XML Information Set recommendation [134] and has become the foundation for most activities of the W3C related to query languages.

Path expressions. The core expressions of XPath are "location steps". A location step specifies where to navigate from the so-called "context node", i.e., the current node of a path traversal. A location step consists of three parts: an "axis", a "node-test", and an optional "predicate". The axis specifies candidate nodes in terms of the tree data model: the base axes `self`, `child`, `following-sibling`, and `following` (selecting the context node, their children, their siblings, or all elements if they occur later in document order, resp.), the transitive and transitive-reflexive closure axes `descendant` and `descendant-or-self` of the axis `child`, and the respective "reverse" (or inverse) axes `parent`, `preceding-sibling`, `preceding`, `ancestor`, and `ancestor-or-self`. Two additional axes, `attributes` and `namespace`, give access to attributes and namespace declarations. Both node-tests and predicates serve to restrict the set of candidate nodes selected by an axis. The node-test can either restrict the label of the node (in case of element and attribute nodes), or the type of the node (e.g., restrict to comment children of an element). Predicates serve to further restrict elements to some neighborhood (which nodes are in the neighborhood of the node selected by an axis and node-test) or using functions (e.g., arithmetic or boolean operators).

Successive location steps are separated by "`/`" to form a path expression. A path expression can be seen as a nested iteration over the nodes selected by each location step. E.g., the path expression `child::book/descendant::name` expresses: *"for each `book` child of the context node select its `name` descendant"*.

XPath compares to generalized path expressions as follows:

- XPath allows navigation in all directions, while generalized path expressions only allow vertical and downwards navigation.

- XPath provides closure axes, but does not allow closure of arbitrary path expressions, e.g. as provided in Lorel.

- XPath has no means for defining variables, as it is intended to be used embedded in a host language that may provide such means. In contrast, Lorel and StruQL offer variables for connecting path expressions, making it possible to specify so-called tree or graph queries. Instead, XPath predicates may contain nested path expressions and thus allow the expression of tree and even some graph queries. However, not all graph queries can be expressed this way. This has been recognized in XPath 2.0 [44], a revision of XPath currently under development at the W3C.

Reverse navigation has been considered for generalized path expressions, cf. [96, 97]). However, it has been shown in [298] that reverse axes do not increase the expressive power of path navigations.

Without closure of arbitrary path expressions, XPath cannot express regular path expressions such as `a.(b.c)*.d` (meaning *"select `d`'s that are reached via one `a` and then arbitrary many repetitions of one `b` followed by one `c`"*) and `a.b*.c`, cf. [266, 265], where also a first-order complete extension to XPath is proposed that can express the second of the above-mentioned path expressions.

Query 1 can only be approximated in XPath as follows:

```
/descendant::book[attribute::type =
/descendant::category[attribute::id = "Essay"]/
descendant-or-self::category/attribute::id]
```

XPath always returns a single set of nodes and provides no construction. Therefore, it is not possible to return authors and their names together with the book.

XPath also has an "abbreviated syntax". In this syntax the above query can more concisely be expressed as:

```
//book[@type = "Essay" or //category[@::id = "Essay"]/
descendant-or-self::category/@id]
```

Query 2 can be expressed in (abbreviated) XPath as:

```
//book[title="Bellum Civile"]
```

XPath returns a set of nodes as result of a query, the serialization being left to the host language. Most host languages consider as results the sub-trees rooted at the selected nodes, as desired by this query. The link to the category is not expressed by means of the XML hierarchy and therefore not included in the result.

Query 3 can be approximated in XPath (assuming we identify "ontology information" with `category` elements):

```
/bookdata//*[name(.) != "translator" and name(.) != "category"]
```

This query returns all descendants of the document element `bookdata` the labels of which (returned by the XPath function `name`) are neither `"translator"` nor `"category"`. While this might at first glance seem to be a convenient solution for Query 3 (the set of nodes returned by the expression indeed does not contain translators and categories), the link between selected `book` nodes and the excluded translators is not removed and in most host languages of XPath the translators would be included as part of their `book` parent.

Queries 4 and 7–9 cannot be expressed in XPath because they all require some form of construction. Aggregations, needed by Query 5, are provided by XPath. Query 5 can be expressed as follows:

```
max(//book[author/name="Julius Caesar"]/year)
```

The aggregation in Query 6 can be expressed analogously. However, Query 6 like Query 1 cannot be expressed in XPath properly due to the lack of construction.

XPath in industry and research. Thanks to XPath's ubiquity in W3C standards (in XML Schema [156], in XSLT [120], in XPointer [189], in XQuery [58], in DOM Level 3), XPath has been adopted widely in industry both as part of implementations of the afore-mentioned W3C standards and in contexts not (yet) considered by the W3C, e.g., for policy specifications. It has also been included in a number of APIs for XML processing in languages for providing easy access to data in XML documents.

XPath has also been deeply investigated in research. *Formal semantics* for (more or less complete) fragments for XPath have been proposed in [363, 298, 181]. Surprisingly, most popular implementations of XPath embedded within XSLT processors exhibit exponential behavior, even for fairly small data and large queries. However, the *combined complexity* of XPath query evaluation has been shown to be P-complete [184, 185]. Various sub-languages of XPath (e.g., forward XPath [298], Core or Navigational XPath [184], [39]) and extensions (e.g., CXPath [265]) have been investigated, mostly with regard to expressiveness and complexity for query evaluation. Also, satisfiability of positive XPath expressions is known to be in NP and, even for expressions without boolean operators, NP-hard [205]. Containment of XPath queries (with or without additional constraints, e.g., by means of a document schema) has been investigated as well, cf., e.g., [374, 147, 280, 331]. Several methods providing efficient implementations of XPath relying on standard relational database systems have been published, cf., e.g., [191, 192, 299].

Currently, the W3C is, as part of its activity on specifying the XML query language XQuery, developing a revision of XPath: XPath 2.0 [44]. See [224] for an introduction. The most striking additions in XPath 2.0 are: (1) a facility for defining variables (using `for` expressions), (2) sequences instead of sets as answers, (3) the move from the value typed XPath 1.0 to extensive support for XML schema types in a strongly typed language, (4) a considerably expanded library of functions and operators [257], and (5) a complete formal semantics [151].

Project pages:

`http://www.w3.org/TR/xpath` for XPath 1.0
`http://www.w3.org/TR/xpath20/` for XPath 2.0

Implementations:

numerous, mostly as part of implementations of XPath host languages or APIs for processing XML (e.g., W3C's DOM Level 3)

Online demonstration:

none (offline XPathTester `http://xml.coverpages.org/ni2001-05-25-a.html`)

XPathLog. XPathLog [270] is syntactically an extension of XPath but its semantics and evaluation are based on logic programming, more specifically F-Logic and FLORID [250]. XPathLog extends the syntax of XPath as follows: (1) variables may occur in path expressions, e.g., `//book[name → N] → B` binds `B` to books and `N` to the names of the books, and (2) both existential and universal quantifiers can be used in Boolean expressions. The data model of XPathLog deviates considerably from XPath's data model: XML documents are viewed in XPathLog as edge-labeled graphs with implicit dereferencing of ID/IDREF references. XPathLog provides means for constructing new or updating the existing XML data, as well as more advanced reactive features such as integrity constraints.

Project page:
    `http://dbis.informatik.uni-goettingen.de/lopix/`

Implementation:
    With the LoPiX system, available from the project page

Online demonstration:
    none

FnQuery. FnQuery [335] is another approach for combining path expressions with logic programming. Attribute lists are used to define a novel representation of XML in Prolog called "field-notation". Data in this representation can then be queried using FnPath: E.g., the expression

```
D^bookdata^book-[^title:’Bellum Civile’, ^year:1992]
```

returns the book with title "Bellum Civile" published in "1990" if the sample data from Section 7.2.2 is bound to `D`. As XPathLog FnQuery allows multiple variables in a path expression. It has been used, e.g., for visualizing knowledge bases [337] and querying OWL ontologies [336].

Project page:
    `http://www-info1.informatik.uni-wuerzburg.de/database/research_seipel.html`

Implementation:
    not publicly available

Online demonstration:
    none

7.3.1.2 The Transformation Language XSLT

XSLT [120], the Extensible Stylesheet Language, is a language for *transforming* XML documents. Transformation is here understood as the process of creating a new XML document based upon a given one. The distinction between querying and transformation has become increasingly blurred as expressiveness of both query and transformation languages increase. Typically, transformation languages are very convenient for expressing selection, restructuring and reduction queries, such as Query 3 above.

XSLT uses an XML syntax with embedded XPath expressions. While the XML syntax makes processing and generation of XSLT stylesheets easier (cf. [368]), it has been criticized as hard to read and overly verbose. Also XPath expressions use a non-XML syntax requiring a specialized parser.

XSLT computations. An XSLT program (called "stylesheet" reflecting the origin of XSLT as part of the XSL project) is composed of one or more transformation rules (called *templates*) that recursively operate on a single input document. Transformation rules are guarded by XPath expressions. In a template, one can specify (1) the resulting shape of the elements matched by the guard expression and (2) which elements in the input tree to process next with what templates. The selection of the elements to process further is done using an XPath expression. If no specific restriction is given, all templates with guards matching these elements are considered, but one can also specify a single (named) template or a group of templates by changing the so-called *mode* of processing. XSLT allows also recursive templates. However, recursion is limited: except for templates constructing strings only, the result of a template is immutable (a so-called *result tree fragment*) and cannot be input for further templates except for literal copies. This means in particular, that no views can be defined in XSLT. Work in [229] shows that XSLT is nevertheless Turing complete, by using recursive templates with string parameters and XSLT's powerful string processing functions.

XSLT 2.0. Recently this and other limitations (e.g., the ability to process only a single input document, no support for XML Schema, limited support for namespaces, lack of specific grouping constructs) have lead to a revision of XSLT: XSLT 2.0 [227]. As with XQuery 1.0, this language is based upon XPath 2.0 [44]. It addresses the above mentioned concerns, in particular adding XML schema support, powerful grouping constructs, and proper views. The XQuery 1.0 and XPath 2.0 function and operator library [257] is also available in XSLT 2.0.

Sample Queries. All example queries can be expressed in XSLT. Query 2 and 5 to 8 are omitted as their solutions are similar enough to solutions shown in the following.

Query 1 can be expressed in XSLT as follows:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<results>
<xsl:apply-templates select="//book[@type =
//category[@id = ’Essay’]/descendant−or−self::category/@id]"/>
</results>
</xsl:template>
<xsl:template match="book">
<result>
<xsl:copy select = "."/>
<xsl:apply-templates select="author|author/name" />
</result>
</xsl:template>
<xsl:template match="author|author/name">
<xsl:copy-of select="." />
</xsl:template>
</xsl:stylesheet>
```

This stylesheet can be evaluated as follows:

- try to match the root node (matched by the guard / of the template in line 3) with the guards of templates in the style-sheet (only first template matches)
- create a `<results>` element and within it try to recursively apply the templates to all nodes matched by the XPath expression in the `select` attribute of the `apply-templates` statement in line 5.
- such nodes are book elements matched by the second template which creates a `<result>` element, makes a shallow copy of itself and recursively applies the rules to the book's author children and their name children.
- for each author or name of an author, copy the complete input to the result.

Aside from templates, XSLT also provides explicit iteration, selection, and assignment constructs: `xsl:for-each`, `xsl:if`, `xsl:variable` among others. Using these constructs one can formulate Query 1 alternatively as follows:

```
<results xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:for-each select="//book[@type = //category[@id = ’Essay’]/
descendant-or-self::category/@id]">
<result>
<xsl:copy select = "."/>
<xsl:for-each select = "author|author/name">
<xsl:copy-of select="." />
</xsl:for-each>
</result>
</xsl:for-each>
</results>
```

The `xsl:for-each` expressions iterate over the elements of the node set selected by the XPath expression in their `select` attribute. Aside from the expressions for copying input this very much resembles the solution for Query 1 in XQuery shown in the following section.

Whereas the first style of programming in XSLT is sometimes referred to as rule-based, the latter one is known as the “fill-in-the-blanks” style, as one specifies essentially the shape of the output with “blanks” to be filled with the result of XSLT expressions. Other programming styles in XSLT can be identified, cf. [225].

Query 3 can be expressed in XSLT as follows:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="@*|node()">
<xsl:copy>
<xsl:apply-templates select="@*|node()"/>
</xsl:copy>
</xsl:template>
<xsl:template match="translator | category" />
</xsl:stylesheet>
```

The first template specifies that for all attributes and nodes, the node itself is copied and their (attribute and node) children are processed recursively. The second template specifies that for translators and category elements, nothing is generated (and their children are *not* processed). Notice that the first template also matches translator and category elements. For such a case where multiple templates match, XSLT uses detailed conflict resolution policies. In this case, the second template is chosen as it is more specific than the first one (for more the details of resolution rules, refer to [120]).

Query 4 can be expressed in XSLT as follows:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<bookdata>
<xsl:apply-template
select="//author[not(name = preceding::author/name)]" />
</bookdata>
</xsl:template>
<xsl:template match="author">
<person>
<name><xsl:value-of select="name" /></name>
<authored>
<xsl:apply-templates
select="//book[author/name=current()/name]" />
</authored>
</person>
</xsl:template>
<xsl:template match="book">
<book>
<xsl:copy-of select="@*" />
<xsl:copy-of select="*[name() != ’author’]" />
</book>
</xsl:template>
</xsl:stylesheet>
```

The preceding axis from XPath is used to avoid duplicates in the result. Also note the use of the current() function in the second template. This function always returns the current node considered by an XSLT expression. Here, it returns the author element last matched by the second template. This function is essentially syntactic sugar to limit the use of variables (cf. solution for Query 9).

Query 9 can be expressed in XSLT as follows:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<results>
<xsl:for-each select="//author">
<xsl:variable name="author" select="." />
<xsl:for-each select="$author/following-sibling::author">
<co-authors>
<name> <xsl:value-of select="$author/name" /> </name>
<name> <xsl:value-of select="current()/name" /> </name>
</co-authors>
</xsl:for-each>
</xsl:for-each>
</results>
</xsl:template>
</xsl:stylesheet>
```

Here, the solution is quite similar to the XQuery solution for Query 9 shown below (but can use in following-sibling axis that is only optionally available in XQuery), as variables and xsl:for-each expressions are used. The solution uses xsl:for-each, as the inner and the outer author are processed differently. A solution without xsl:for-each is possible but requires parameterized templates and named or grouped templates:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<results>
<xsl:apply-template select="//author" />
</results>
</xsl:template>
<xsl:template match="author">
<xsl:apply-template select="following-sibling::author"
mode="co-author">
<xsl:with-param name="first-co-author" select="." />
</xsl:apply-templates>
</xsl:template>
<xsl:template match="author" mode="co-author">
<xsl:param name="first-co-author" />
<co-authors>
<name> <xsl:value-of select="$first-co-author/name" /> </name>
<name> <xsl:value-of select="name" /> </name>
</co-authors>
</xsl:template>
</xsl:stylesheet>
```

Note that for clarity neither of these solutions avoids duplicates if two persons are co-authors of multiple books.

XSLT in industry and academia. XSLT has been the first W3C language for transforming and querying XML and thus has been adopted quickly and widely. A multitude of implementations exist (e.g. as part of the standard library for XML processing in Java) as well as good practical introductions (e.g.., [354, 225]).

Research on XSLT has not received the same attention that XPath and XQuery have, in particular not from the database community. A more detailed overview of research issues on XSLT and its connection to reactive rules is given in [23], here only some core results are outlined: Formal semantics for (fragments of) XSLT have been investigated in [231, 53]. [229] gives a proof showing that XSLT is Turing complete.

Analysis of XSLT is examined in [150], which proposes four analysis properties and presents an analysis method based on the construction of a template association graph, which conservatively models the control flow of the stylesheet. There is also an important line of theoretical research with regard to analysis of the behaviour of XSLT. Work in [283] presents a theoretical model of XSLT and examines a number of decision questions for fragments of this model. Work in [264] examines the question of whether the output of an XML transformation conforms to a particular document type. Type checking is also addressed in [359].

Efficient evaluation of XSLT programs is also an important topic. In [216, 248], translations to SQL are considered. Work in [362] describes incremental methods for processing multiple transformations. Work in [330] proposes a lazy evaluation for XSLT programs, while [226] describes optimizations based on experiences from the widely used XSLT processor Saxon. Other specific techniques for optimizing XSLT programs and evaluation are described in [187, 360, 149, 198]. Further engineering aspects of XSLT programs have also received attention, namely transformation debugging [22] and automatic stylesheet generation [300, 368].

Project page:
`http://www.w3.org/Style/XSL/`

Implementation:
very numerous, see project page

Online demonstration:
none

Fxt. *fxt* [45], the *functional XML transformer*, is a transformation language similar to XSLT, in particular with respect to its syntax. However, instead of XPath expressions *fxt* uses *fxgrep* patterns that are based on an expressive grammar formalisms and can be evaluated very efficiently (cf. [46]). *Fxt*'s computation model is also more restricted than that of XSLT due to the lack of named templates.

Project page:
`http://atseidl2.informatik.tu-muenchen.de/~berlea/Fxt/`

Implementation:
available from the project page

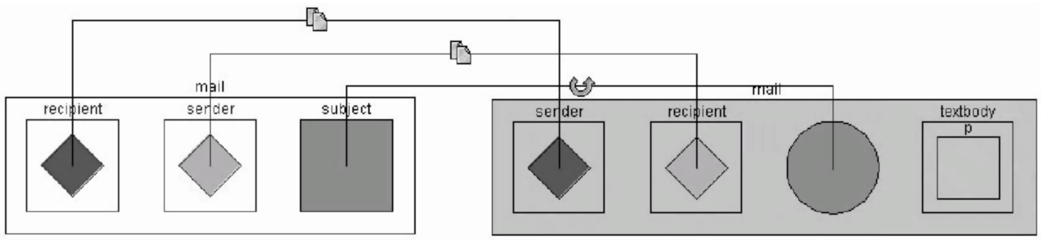
Online demonstration:
none

VXT. VXT [307] is a visual language and interactive environment for specifying transformations of XML documents. It is based on the general purpose transformation language Circus[5]: Whereas most other XML query languages employ some form of graph-shaped visualization for both data and queries, VXT uses treemaps [217] for representing hierarchies: the nesting of the elements in the document is reflected by nested of nodes. As XSLT, VXT uses rules to specify transformations. A rule consists in treemap representation of the queried data and the constructed data. The two representations are linked by various typed edges indicating, e.g., the copying of a matching node or its content, cf. 7.3

Project page:
none

[5] `http://www.xrce.xerox.com/solutions/circus.html`

Figure 7.3 Treemap representation of a VXT rule
([307], © ACM Press)



Implementation:
not publicly available

Online demonstration:
none

7.3.1.3 The Query Language XQuery

Shortly before the publication of the final XPath 1.0 and XSLT 1.0 recommendations, the W3C launched an activity towards specifying an XML query language. In contrast to XSLT, this query language aims at a syntax and semantics making it convenient for database systems. Requirements and use cases for the language have been given in [255, 108, 107]. A number of proposals, e.g., XQL and Quilt, have been published in answer to this activity, each with varying influence on XQuery [58], the language currently under standardisation at the W3C:

XQL [325, 323] notably influenced the development of XPath. Although XQL did not consider the full range of XPath axes, some language features that have not been included in XPath, e.g., existential and universal quantifiers and an extended range of set operations, are under reconsideration for XPath 2.0.

Quilt [110] is in spirit already close to the current version of XQuery, mainly lacking the extensive type system developed by the W3C's XML query working group. It can be considered the predecessor of XQuery.

Although the development and standardisation of XQuery [58] is not completed, XQuery's main principles have been unchanged during at least the last two of its four years of development. In many respects, it represents the "state-of-the-art" of navigational XML query languages.

XQuery Principles. At its core, XQuery is an extension of XPath 2.0 adding functionalities needed by a "full query language". The most notable of these functionalities are:

- *Sequences.* Where in XPath 1.0 the results of path expressions are node sets, XQuery and XPath 2.0 use sequences. Sequences can be constructed or result from the evaluation of an XQuery expression. In contrast to XPath 1.0, sequences cannot only be composed of nodes but also from atomic values, e.g., `(1, 2, 3)` is a proper XQuery sequence.

- *Strong typing.* Like XPath 2.0, XQuery is a strongly typed language. In particular, most of the (simple and complex) data types of XML Schema are supported. The details of the type system are described in [151]. Furthermore, many XQuery implementations provide (although it is an optional feature) static type checking.

- *Construction, Grouping, and Ordering.* Where XPath is limited to selecting parts of the input data, XQuery provides ample support for constructing new data. Constructors for all node types as well as the simple data types from XML Schema are provided. New elements can be created either by so-called direct element constructors (that look just like XML elements) or by what is referred to as computed element constructors, e.g. allowing the name of a newly constructed element to be the result of a part of the query. For examples on these constructors, see the implementations for Query 1 and 3 below.
- *Variables.* Like XPath 2.0, XQuery has variables defined in so-called FLWOR expressions. A FLWOR expression usually consists in one or more `for`, an optional `where` clause, an optional `order by`, and a `return` clause. The `for` clause iterates over the items in the sequence returned by the path expression in its `in` part: `for $book in //book` iterates over all books selected by the path expression `//book`. The `where` clause specifies conditions on the selected data items, the `order by` clause allows the items to be processed in a certain order, and the `return` clause specifies the result of the entire FLWOR expression (often using constructors as shown above). Additionally, FLWOR expressions may contain, after the `for` clauses, `let` clauses that also bind variables but without iterating over the individual data items in the sequence bound to the variable. FLWOR expressions resemble very much XSLT's explicit iteration, selection, and assignment constructs described above.
- *User-defined functions.* XQuery allows the user to define new functions specified in XQuery (cf. implementation of Query 3 below). Functions may be recursive.
- *Unordered sequences.* As a means for assisting query optimization, XQuery provides the `unordered` keyword, indicating that the order of elements in sequences that are constructed or returned as result of XQuery expressions is not relevant. E.g., `unordered{for $book in //book return $book/name}` indicates that the nodes selected by `//book` may be processed in any order in the `for` clause and the order of the resulting name nodes also can be arbitrary (implementation dependent). Note that inside unordered query parts, the result of any expressions querying the order of elements in sequences such as `fn:position`, `fn:last` is non-deterministic.
- *Universal and existential quantification.* Both XPath 2.0 and XQuery 1.0 provide `some` and `all` for expressing existentially or universally quantified conditions (see implementation of Query 9 below).
- *Schema validation.* XQuery implementations may (optionally) provide support for schema validation, both of input and of constructed data, using the `validate` expression.
- *Full host language.* XQuery completes XPath with capabilities to set up the context of path expressions, e.g., declaring namespace prefixes and default namespace, importing function libraries and modules (optional), and (again optionally) providing flexible means for serialization that are in fact shared with XSLT 2.0 (cf. [228]).

In at least one respect, XQuery is more restrictive than XPath: not all of XPath's axes are mandatory, `ancestor`, `ancestor-or-self`, `following`,
`following-sibling`, `preceding`, and `preceding-sibling` do not have to be supported by an XQuery implementation. This is, however, no restriction to XQuery's expressiveness, as expressions using reverse axes (such as `ancestor`) can be rewritten, cf. [298], and the "horizontal axes", e.g., `following` and `following-sibling`, can be replaced by FLWOR expressions using the « and » operators that compare two nodes with respect to their position in a sequence.

For a formal semantics for XQuery 1.0 (and XPath 2.0) see [151]. Comprehensive but easy to follow introductions to XQuery are given in, e.g., [223, 80].

Sample Queries. All nine sample queries can be expressed in XQuery. In the following, an expression of Query 2 is omitted because it can be expressed as a simplification of the XQuery expression of Query 1 given below. Query 5 can be expressed as for XPath, cf. above. Expressions of Query 8 and 9 are similar. Since the expression for Query 9 in XQuery exhibits an interesting anomaly, it is given below and no expression for Query 8 is given.

Query 1 can be expressed in XQuery as follows (interpreting the phrase "an essay" as a book with type attribute equal to the id of the category "Essay" or one of its sub-categories represented as descendants in the XML structure):

```
<results> {
let $doc := doc("http://example.org/books")/bookdata
let $sub-of-essay :=
$doc//category[@id="Essay"]/descendant-or-self::category
for $book in $doc//book
where $book/@type = $sub-of-essay/@id
return
<result>
{ $book }
{ $book/author }
{ $book/author/name }
</result> }
</results>
```

Note the use of the let clause in line 2: the sequence of all sub-categories of the category with id "Essay" including that category itself (we use the reflexive transitive axis descendant-or-self) is bound to the variable. However, in contrast to a for expression, this sequence is not iterated over. Instead of the where clause in line 4 a predicate could be added to the path expression in line 3 resulting in the expression $doc//book[@type = $sub-of-essay/@id].

Query 3 requires structural recursion over the tree, while constructing new elements that are identical to the ones encountered, except omitting translator and category nodes. The following implementation shows the use of a user-defined, recursive function that copies the tree rooted at its first parameter $e, except all nodes in the sequence given as second parameter.

```
declare function
local:tree-except($e as element(),
$exceptions as node()*) as element()*
{
element {fn:node-name($e)} {
$e/@* except $exceptions, (: copy the attributes :)
for $child in $element/node() except $exceptions
return
if $child instance of element()
(: for elements process them recursively :)
local:tree-except($section)
else (: others (text, comments, etc. copy :)
$child
}
};
document {
let $doc := doc("http://example.org/books")/bookdata
let $exceptions := $doc//translator union $doc//category
local:tree-except($doc, $exceptions)
}
```

Note the typing of the parameters: the first parameter is a single element, the second, a sequence of nodes and the function returns a sequence of elements. In the main part of the query, the document constructor is used to indicate that its content is to be the document element of the constructed tree.

Query 4 can be expressed in XQuery as follows:

```
<bookdata> {
let $a := doc("http://example.org/books")//author
for $name in distinct-values($a/name)
return
<person>
<name> { $name } </name>
<authored
{
for $b in doc("http://example.org/books")//book
where some $ba in $b/author
satisfies $ba/name = $name
return
<book> { $b/@*, $b/* except $b/author } </book>
}
</authored
</person>
}
</bookdata>
```

This implementation is in fact similar to the implementation of use case XMP-Q4 in [107] and exhibits two noteworthy functionalities: (1) The use of `distinct-value` in line 3 to avoid duplication in the result, if an author occurs multiple times in the document. (2) The use of an existentially quantified condition in lines 10–11, to find books where some (read: at least one) of the authors have the same name as the currently considered author.

Using aggregation expressions (see lines 8 and 10), Query 6 can be expressed in XQuery as follows:

```
<results> {
let $doc := doc("http://example.org/books")/bookdata
for $category in $doc//category[@id="Essay"]//category
return
<category>
{ $category/@id }
<average-number-of-authors>{
fn:avg(for $book in $doc//book
where @type = $category/@id
return fn:count($book/author))
}
</average-number-of-authors>
</category>
}
</results>
```

Combining data can be expressed in a very compact manner in XQuery, as the following expression of Query 7 shows:

```
<book>
{ for $book in doc("http://example.org/books")//book
where title="Bellum Civile" and author/name="Julius Caesar"
return ($book/@*, $book/*)
}
{
for $book in doc("http://example.org/books")//book
where @id="bellum_civile"
return ($book/@*, $book/*)
}
</book>
```

Query 9 can be expressed in XQuery as follows:

```
<results>
{ let $doc := doc("http://example.org/books")
for $book in doc("http://example.org/books")//book
for $author in $book/author
for $co-author in $book/author
where $author << $co-author
return
<co-authors>
<name> { $author/name } </name>
<name> { $co-author/name } </name>
</co-authors>
}
</results>
```

This implementation does not treat the case where two authors co-authored multiple books. In this case, duplicates are created by the above solution. To avoid this the following refinement uses the before operator « in combination with a negated condition, for specifying that only such pairs of authors should be considered, where there is no book that occurs prior to the currently considered one and which is also co-authored by the current pair of authors:

```
<results>
{ let $doc := doc("http://example.org/books")
for $book in doc("http://example.org/books")//book
for $author in $book/author
for $co-author in $book/author
where $author << $co-author and not(
some $pb in doc("http://example.org/books")//book
satisfies ($pb << $book and
$pb//author/name = $author/name and
$pb//author/name = $co-author/name))
return
<co-authors>
<name> { $author/name } </name>
<name> { $co-author/name } </name>
</co-authors>
}
</results>
```

XQuery in industry and research. From the very start, XQuery's development has been followed by industry and research with equal interest (for reports on the challenges and decisions during this process see, e.g., [154, 157]). Even before the development has finished, initial practical introductions to XQuery have been published, e.g., [223, 80]. Industry interest is also visible in the simultaneous development of standardized XQuery APIs, e.g., for Java [155], and numerous implementations, both open source (e.g., Galax [161]) and commercial (BEA [167], IPSI-XQ [158]). Aside from these main-memory implementations, one can also find streamed implementations of XQuery (e.g., [235, 34]) where the data flows by as the query is evaluated. First results on implementing XQuery on top of standard relational databases (e.g., [139, 193]) indicate that this approach leads to very efficient query evaluation if a suitable relational encoding of the XML data is used. For more implementations, see the XQuery project page at the W3C and the proceedings of the first XIME-P workshop on "XQuery Implementation, Experience and Perspectives"[6].

It is intuitively clear that XQuery is Turing complete since it provides recursive functions and conditional expressions. A formal proof of the Turing-completeness of XQuery is given in [229]. Efficient processing and (algebraic) optimization of XQuery, although acknowledged as crucial topics, have not yet been sufficiently investigated. First results are presented, e.g., in [377, 113, 376, 115, 268, 146, 353]. Moreover, techniques for efficient XPath evaluation, as discussed above, can be a foundation for XQuery optimization.

Beyond querying XML data, it has also been suggested to use XQuery for data mining [367], for web service implementation [301], for querying heterogeneous *relational* databases [370], for access control and policy descriptions [285], for synopsis generation [132], and as the foundation of a visual XML query language (XQBE) [19], of a XML query language with full-text capabilities [14, 13], and of an update [322, 81, 109] and reactive [64] language for XML.

Project page:

`http://www.w3.org/XML/Query`

[6] http://www-rocq.inria.fr/gemo/Gemo/Projects/XIME-P/

Implementations:

widely implementated (more than 30 implementations), a list of implementations is available at the project page

Online demonstrations:

several, e.g.: `http://www.oakleaf.ws/xquery/xquerydemo.aspx`
`http://oasys.ipsi.fhg.de/xquerydemo/`
`http://131.107.228.20/xquerydemo/demo.aspx`

7.3.2 Research Prototypes: The Positional Approach to XML Querying

7.3.2.1 Characteristics of the Positional Approach.

The languages discussed in the following all take the *positional approach* for locating data in an XML document. This approach is often derived from logic or functional programming where patterns are used to specify the position of interesting data inside larger structures.

Essentially, positional languages use expressions that mimic the data to be queried. This allows tree- or graph-shaped *queries* to be expressed very similar to tree- or graph-shaped *data* (as "examples" of the data to be queried, cf. [379]), whereas navigational languages do not provide this close correspondence. However, many languages in this sections (e.g., UnQL, TQL, and Xcerpt) do actually use path expressions mostly as convenient shorthands for parts of queries that are shaped like a single path.

Languages using this "query-by-example" style for queries mostly fall into two categories: (a) query languages influenced by logic or functional programming (UnQL, XML-QL, XMAS, XML-RL, TQL) and (b) visual query languages or visual interfaces for textual query languages (XML-GL, BBQ, and X^2's visual query interface).

7.3.2.2 UnQL.

UnQL [92,91,93] (the *Unstructured Query Language*) is a query language originally developed for querying semistructured data and nested relational data-bases with cyclic structures. It has later been adopted to querying XML, but the origins are still apparent in many language properties (for example, UnQL has a non-XML syntax that is very similar to OEM's syntax and does not support querying or construction of ordered data).

The evaluation model and core language of UnQL is based upon structural recursion over labeled trees. It provides both a functional-style language for expressing recursions over trees, cf. [92] and a more approachable surface syntax.[7]

The following expression uses functional style pattern matching for selecting all books in a tree.

```
fun f1(T1 u T2) = f1(T1) u f1(T2)
| f1({ L <-- T }) = if L = book then {result <-- book <-- T} else f1(T)
| f1({}) = {}
| f1(V) = {}
```

UnQL's *surface syntax* uses *query patterns* and *construction patterns* and a query consists of a single `select ... where ...` or `traverse` rule that separate construction from querying. Queries may be nested, in which case the separation of querying and construction is abandoned.

Query 1 can be expressed in UnQL as

[7] The syntax from [92,91] is used and not the slightly differing syntax in [93].

```
select { results <-- {
select { result <-- { Book,
select { author <-- {
author <-- Author,
authorName <-- Name
} }
where { author <-- \Author } -→ Book,
{ name <-- \Name } -→ Author
where { book <-- \Book } -→ Bib
where bookdata <-- Bib -→ DB
```

The ← scopes a query pattern, i.e., it specifies that the left-hand query pattern is to be found in bindings for the right-hand variable. The ⇒ operator is the direct edge traversal operator. E.g., `book ⇒ author` specifies that `author` is a direct child of `book` in the XML document. Recursive traversals can be specified using regular path expressions including regular expressions over labels. E.g., `_*` traverses over arbitrary many elements with any label, `[^book]*` over arbitrary many elements with any label except `book`.

UnQL also provides `traverse` clauses for reduction and restructuring queries like Query 3:

```
traverse DB given X
case translator <-- _ then X := {}
case category <-- _ then X := {}
case \L <-- _ then X := {l <-- X}
```

This query is evaluated by traversing the tree in the database and matching recursively each element against the three `case` expressions. All elements except translators and categories are copied to the newly constructed tree, structured as in the input data.

UnQL is probably the first language to propose a pattern-based querying (albeit with subqueries instead of rule chaining) for semistructured data (including XML).

Evaluation and optimization of UnQL has been investigated in [91,93]. UnQL's evaluation is founded in graph simulation, see [93]. [91] shows that all queries expressible in UnQL can be evaluated in PTIME. This is true even for queries against cyclic graph data (e.g. XML documents using cyclic ID/IDREF references). This efficiency is reflected by UnQL's expressiveness: on trees encoding relational or nested relational databases, UnQL is exactly as expressive as relational or nested relational algebra, resp.

Project page:
`http://www.research.att.com/~suciu/unql-home.html`[8]

Implementation:
available from the project page

Online demonstration:
none

7.3.2.3 XML-QL.

XML-QL [144,145] is a pattern- and rule-based query language for XML developed specifically to address the W3C's call for an XML query language (that resulted in the development of XQuery). Like UnQL, it uses *query patterns* (called *element patterns* in [144]) in a `WHERE` clause. Such patterns can be augmented by variables for selecting data. The result of a query is specified as a *construction patterns* in the `CONSTRUCT` clause. An XML-QL query always consists of a single `WHERE-CONSTRUCT` rule, which may be divided into several (nested) subqueries.

Query 1 can be expressed in XML-QL as follows:

[8] Not accessible at the time of writing.

```
WHERE
<bookdata>
<book>
</> ELEMENT_AS $b
</>
CONSTRUCT
<results>
<result>
$b
WHERE <author>
<name> $n </>
</> ELEMENT_AS $a
CONSTRUCT $a
$n
</>
</>
```

Variables are preceded in XML-QL by `$`. Note how the grouping of authors with their books is expressed using a nested query. Also note the tag minimization (end tags abbreviated by `</>` as in SGML), e.g., in line 4 and 5. In line 4, the variable `$b` is restricted to data matching the pattern in lines 3 and 4. Such "pattern restrictions" are indicated in XML-QL using the `ELEMENT_AS` keyword.

One of the main characteristics of XML-QL is that it uses query patterns containing multiple variables that may select several data items at a time instead of path selections that may only select one data item at a time. Furthermore, variables are similar to the variables of logic programming, i.e. "joins" can be evaluated over variable name equality. Since XML-QL does not allow one to use more than one separate rule, it is often necessary to employ subqueries to perform complex queries.

Query 6 cannot be expressed in XML-QL due to lack of aggregation, in particular structural aggregation (e.g., counting the number of children of an element). The following query returns all books classified in a sub-category of "Novel":

```
WHERE
<book type=$Sub>
</> ELEMENT_AS $b,
<category id=’Novel’>
<category* id=$Sub>
</>
</>
CONSTRUCT $b
```

As discussed, above joins are simply expressed by repeated occurrences of the same variable (lines 2 and 5). In line 5 a further feature of XML-QL is shown: instead of element labels one can use regular path expressions in patterns.

Transformation queries such as Query 2, where the output closely resembles the input except for some rather localized changes (e.g., omission of elements or changing labels), cannot in general be expressed in XML-QL.

Also XML-QL does not provide any means for testing the non-existence of elements and therefore cannot express queries such as "Return all books that have no translator.".

No results on complexity or expressiveness of XML-QL have been published.

Project page:
`http://www.research.att.com/~mff/xmlql/doc/`

Implementation:
available from the project page

Online demonstration:
none

7.3.2.4 XMAS.

XMAS [251], the *XML Matching And Structuring language* is an XML query language developed as part of MIX [30] and builds upon XML-QL. Like XML-QL, XMAS uses *query patterns* and *construction patterns*, and rules of the form CONSTRUCT ...WHERE .... However, XMAS extends XML-QL in that it provides a powerful *grouping construct*, instead of relying on subqueries for grouping data items within an element.

Query 1 can be expressed in XMAS as follows:

```
WHERE
<bookdata>
$B: <book>
$A: <author>
<name> $N </name>
</>
</>
</>
CONSTRUCT
<results>
<result>
$B
<book-author>
$A
<name> $N </name>
</> {$A,$N}
</> {$B}
</>
```

Here, one can observe the two main syntactic differences to XML-QL: (1) In XMAS, grouping is expressed by enclosing the variables on whose bindings the grouping is performed in curly braces and attaching them to the end of the subpattern that specifies the structure of the resulting instances. In the above example, a result element is created for every instance of $B (indicated by {$B} after the closing tag of the element result). Within every such result element, all authors of a book (indicated by {$A}) are collected nested in book-author elements (the book-author element is necessary for grouping variables are allowed only after closing tags or single variables in XMAS).

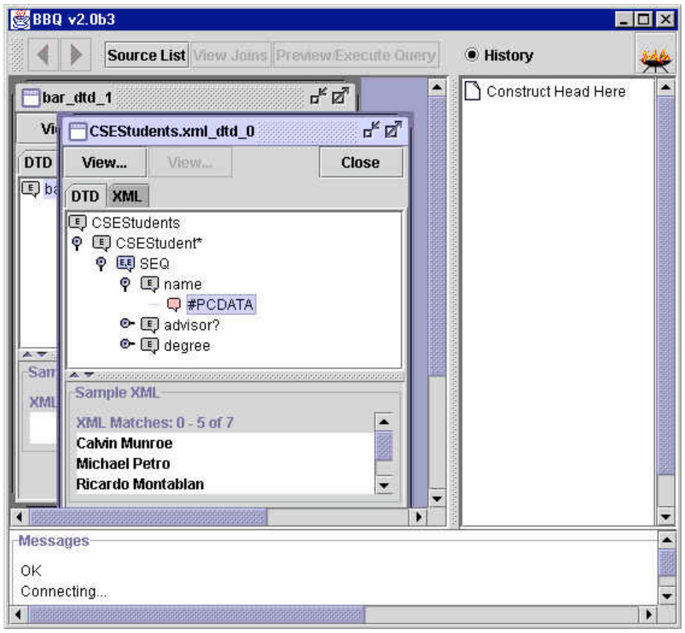
(2) XMAS also provides a more compact syntax for *pattern restrictions* that allow one to restrict the admissible bindings of a variable as seen in line 3 ($B in front of the subpattern instead of XML-QL's ELEMENT_AS $B at the end).

Grouping queries can be specified even more concisely by using "implicit collection labels": instead of specifying the grouping variables explicitly, all variables nested inside square brackets are considered grouping variables for that grouping, unless there is another grouping (i.e., block enclosed by square brackets) closer to the variable occurrence. Using implicit collection labels, Query 1 can be expressed as:

```
WHERE
<bookdata>
$B: <book>
$A: <author>
<name> $N </name>
</>
</>
</>
CONSTRUCT
<results>
[<result>
$B
[<book-author>
$A
<name> $N </name>
</book-author>]
</>]
</>
```

No results on complexity or expressiveness of XMAS have been published.

Figure 7.4 Screenshot of BBQ's query editor
([284], © Kluwer, B.V.)



CSEStudents
CSEStudent*
SEQ
name
#PCDATA
advisor?
degree

XML Matches: 0 - 5 of 7
Calvin Munroe
Michael Petro
Ricardo Montablan

OK
Connecting...

BBQ [284] is a visual interface for XMAS that allows browsing of XML data as well as authoring of XMAS queries based on a DTD of the data to be queried. Figure 7.4 shows the two-pane query editor with a query pattern on the left and an (empty) construct pattern at the right.

Project page:
http://www.db.ucsd.edu/projects/MIX/

Implementation:
publicly available only as part of the BBQ online demonstration

Online demonstration:
using BBQ http://www.db.ucsd.edu/Projects/MIX/BBQ_User_Interface.html

7.3.2.5 XML-RL.

XML-RL [249] is a a pattern-based query language based on logic programming. Patterns are expressed by terms that may contain logic variables and may be partly abbreviated with a path syntax similar to abbreviated XPath. An XML-RL query program consists of one or more rules denoted by $A \Leftarrow L_1, \dots, L_n$ where A is used for construction and $L_1, \dots, L_n$ are query pattern. Rules may interact via rule chaining and it is possible to use recursion.

Query 1 can be expressed in XML-RL as follows:

```
/results/result: (book:$b, {author: $a}, {authorName: $n})
⇐
```

```
(file:bib.xml)
/bookdata/book: $b(author: $a(name:$n))
```

The URL in line 3 defines the input data for the query. Analogously it is also possible to give an URL in the construct part of the query (line 1). Notice the curly brackets in line 1. They specify, that authors and author names are to be grouped by book.

XML-RL does not provide specific support for transformation queries such as Query 3, but they can be solved using recursive rules.

Query 6 can be expressed in XML-RL.

```
/results/result: ($i, avg-number-of-authors: $avg)
⇐
(file:bib.xml)
/bookdata/category: (@id: Writing, category//category/@id: $i),
(file:bib.xml)
/bookdata/book: #b (@type: $i, author: #a),
$avg = count(#a) ÷count(#b) ;
/bookdata/category: (@id: Writing, category/@id: $i),
(file:bib.xml)
/bookdata/book: #b (@type: $i, author: #a),
$avg = count(#a) ÷count(#b)
```

This rule has two alternative query expressions (separated as in Prolog by `;`) but only a single head. The first alternative covers the case of indirect sub-categories of "Writing", the second the case of direct ones. In both cases, the `id` attribute of a category is selected and joined with the `type` attribute of books. The books are collected in the *list* variable `#b`, all their authors in the list variable `#a`. Finally, the average number of authors per publication in that sub-category is computed by dividing the number of elements in the two lists.

No results on complexity or expressiveness of XML-RL have been published.

Project page:
none

Implementation:
not publicly available

Online demonstration:
none

7.3.2.6 TQL.

TQL [133, 99] is an XML query language based upon ambient logic [100], a modal logic conceived for describing the structural and computational properties of distributed and mobile computation. Ambient logic uses, for the structural descriptions at least, a logic of labeled trees and is thus a reasonable foundation for an XML query language.

[99] describes a representation of XML documents in ambient logic, called "information trees": XML is considered an edge-labeled graph. No distinction between attributes and elements is considered. Also the order of elements in an XML document is not preserved.

Based upon this data structure, TQL queries are specified as `from ...select` rules. Query and construction are separated (except for grouping queries that are, as in XML-QL and UnQL, expressed using nested queries), the query is specified in the `from` clause, the construction in the `select` clause. TQL programs consist of a single such rule. Instead of chaining rules, recursion is provided by a special recursion operator `rec` similar to the minimal and maximal fix point operators in modal logic. The following expression (taken from [99]) can be used as a condition in `from` clauses and test, recursively, whether a tree is binary:

```
rec $Binary. 0 Or (%[$Binary] | %[$Binary])
```

Variables are indicated in TQL using `$`. The expression `%[$Binary]` matches elements with arbitrary label (indicated by the wild card `%`) and satisfying the condition specified in square brackets, viz. to be binary trees.

Query 1 can be expressed in TQL as follows (assuming `$Bib` is bound to the sample data from Section 7.2.2:

```
from $Bib |= .bookdata[ .book [ $Book ] ]
select
results [ result [
book [ $Book ]
| from $Book |= .author [
$Author And .name [$Name] ]
select
author-and-name [ author [ $Author ], name [ $Name
] ]
]
]
```

As stated above, grouping queries are expressed using nested queries. Notice, how in line 1 (and in line 6) the `$Book` (`$Author`) variables are bound to the sub-tree reached by a matching `book` (`author`) edge.

TQL provides a rich path syntax for abbreviating path-shaped queries. E.g., the expression

```
from $Bib |= .bookdata.%*.category[!.id[Writing] | .category*.label[$Label]
select $Label
```

returns the value of all labels reachable over arbitrary many category edges (`.category*`) from a category that may occur at any depth (`.%*`) and has no `id` with value "Writing".

In [99], it is claimed that TQL is particularly well suited for testing integrity constraints or schema validation, as it provides full boolean expressions including negation, existential, universal quantification, and (structural) recursion with the `rec` operator.

Project page:
`http://www.di.unipi.it/~ghelli/tql/`

Implementation:
available from the project page

Online demonstration:
none

7.3.2.7 Xcerpt.

Xcerpt [88,87,329,41,328] is a query language designed after principles given in [84] for querying both data on the "standard Web" (e.g., XML and HTML data) and data on the Semantic Web (e.g., RDF, Topic Maps, etc. data). This Section addresses using Xcerpt on the "standard Web", Section 7.4.6, on the Semantic Web.

Xcerpt is "data versatile", i.e. the same Xcerpt query can access and generate, as answers, data in different Web formats. Xcerpt is "strongly answer-closed", i.e. it not only allows one to construct answers in the same data formats as the data queries like, e.g., XQuery [108], but also allows further processing of the data generated by this same query program. Xcerpt's queries are pattern-based and allow to incompletely specify the data to retrieve, by (1) not explicitly specifying all children of an element, (2) specifying descendant elements at indefinite depths (restrictions in the form of regular path expressions being possible), and (3) specifying optional query parts. Xcerpt's evaluation of incomplete queries is based on a novel unification algorithm called "simulation unification" [89,90]. Xcerpt's processing of XML documents is graph-oriented, i.e., Xcerpt is aware of the reference mechanisms (e.g., ID/IDREF attributes and links) of XML. Xcerpt is rule-based. An Xcerpt rule expresses how data queried can be re-assembled into new data items. One might say that an Xcerpt rule corresponds to an SQL *view*. Xcerpt allows both traversal of cyclic documents and recursive rules, termination being ensured by so-called memoing, or tabling, techniques. Xcerpt rules can be chained forward or backward, backward chaining being the processing of choice for the Web. Indeed, if rules can, like Xcerpt's rules, query any Web site, then a forward processing of rule-based programs could require starting a program's evaluation at all Web sites. Xcerpt is inspired from Logic Programming. However, since it does not offer backtracking as a programming concept, Xcerpt can also be seen "set-oriented functional".

All of the queries from Section 7.2.3 can be expressed in Xcerpt. In the following, solutions for Query 2, 5, 7, and 8 are omitted as they are similar to other solutions shown.

Query 1 can be expressed in Xcerpt as follows:

```
GOAL
results [
all result [
var Book,
all var Author,
all var AuthorName
]
]
FROM
bookdata {{
var Book →book {{
var Author →author {{
name [ var AuthorName ] }}
}}
}}
END
```

As stated above, Xcerpt rules allow a separation of construction and querying. In the query part (enclosed by `FROM` and `END`), a pattern of the requested data is specified: a `bookdata` element with a `book` child (associated with the variable `Book` using the "pattern restriction" operator →) that in turn has an `author` child (bound to the variable `Author`) with a `name` child whose content is bound to the Variable `AuthorName`. Notice the use of double curly braces in line 10, indicating an incomplete, unordered pattern. A matching `bookdata` element may have additional children not specified in the query and the order among the children is irrelevant for the query. Square brackets as in line 13 and in the construct part (between `GOAL` and `FROM`) specify that the order of the children matters. Single brackets specify that the pattern is complete. Note that incomplete query patterns might result in several alternative variable bindings.

Similar to XMAS, Xcerpt allows to group answers using the constructs `all` and `some`. Intuitively, `all t` collects all possible different instances of the subexpression `t` that might result from alternative variable bindings. As shown in the example above, grouping constructs may also be nested. In the example above, the construct term creates a `result` subterm for each alternative binding of `Book`, and within each such `result` subterm, it groups all authors and authornames associated with that particular book.

In general, an Xcerpt program may contain multiple rules, as shown in the following solution for Query 3:

```
GOAL
var Result
FROM
transform [ bookdata {{ }}, result [ var Result ] ]
END
CONSTRUCT
```

```
transform [ var Element, result [ ] ]
FROM
desc var Element →/translator|category/
END
CONSTRUCT
transform [ var Element, result [ var Label [ all var Child ] ] ]
FROM
and {
desc var Element →var Label [[ var Child ]]
where {
and { var Label != "translator", var Label != "category }
},
transform [ var Child, result [ var ChildTransformed ] ]
}
END
```

Xcerpt rules come in two flavors: `GOAL ... FROM ... END` and `CONSTRUCT ... FROM ... END`. The first may only occur once in a program, specifies the ultimate result of the entire program similar to Prolog goals, and does not participate in rule chaining. The latter form is used for all other rules.

Here, the two lower rules transform (recursively) an input element as specified in the query: if it is a translator or a category the result of the transformation is empty, otherwise the children of the element are recursively transformed and the result of these transformations is used to reconstruct the structure of the input data.

Notice the use of the `desc` operator in lines 10 and 17 indicating a pattern that is incomplete in depth. Also notice the use of a `where` clause in line 18 to restrict matches to elements that are neither translators nor categories. In line 17, a *label variable* is used: whereas the variable `Element` is bound to the entire element matched by the pattern, `Label` is bound to the label of the element, i.e., a string such as "book".

Query 4 can be expressed in Xcerpt as follows:

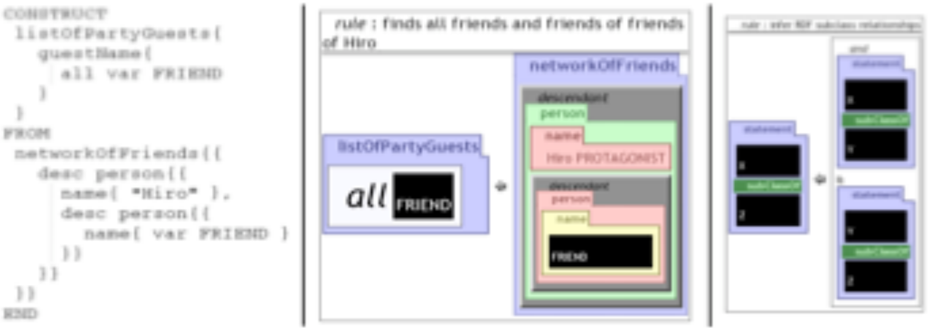
```
GOAL
bookdata [
all person [
name [ var Name ],
authored [
all book [
all var NonAuthorChildren
] group by { var Book }
]
]
]
FROM
bookdata {{
desc var Book →book [[
author {{ name [ var Name ] }},
var NonAuthorChildren →!/author/ {{ }}
]]
}}
END
```

In the query part all books (at any depth) are selected together with the names of their authors and non-author children (notice the use of a negated regular expression on the label for the non-author children). For each name of an author, a `person` element is constructed (note the position of the `all` in line 3) containing the name and an `authored` element. In the author element all books for that author are nested again using `all` with a `group by` clause for explicitly naming the grouping variable.

Query 6 can be expressed in Xcerpt as follows:

```
GOAL
results [
all category [
attributes [ id [ var ID ] ],
average-number-of-authors [
div( count( all var Author ), count( all var Book ) )
]
]
```

Figure 7.5 Xcerpt and visXcerpt representation of a query



```
CONSTRUCT
 listOfPartyGuests[
   guestName[
     all var FRIEND
   ]
 ]
FROM
 networkOfFriends{{
   desc person{{
     name[ "Hiro" ],
     desc person{{
       name[ var FRIEND ]
     }}
   }}
 }}
END
```

```
]
FROM
bookdata {{
desc category {{ attributes {{ id [ var ID ] }} }},
desc var Book →book {{
attributes {{ type [ var ID ] }},
desc var Author →author {{ }}
}}
}}
END
```

The average number of authors is calculated in line 6 using the structural aggregation function `count` over all books and authors for a category. In typical logic-programming style, the join between the `id` attribute of categories and the `type` attribute of books is expressed by repeating the same variable.

Query 9 can be expressed in Xcerpt as follows:

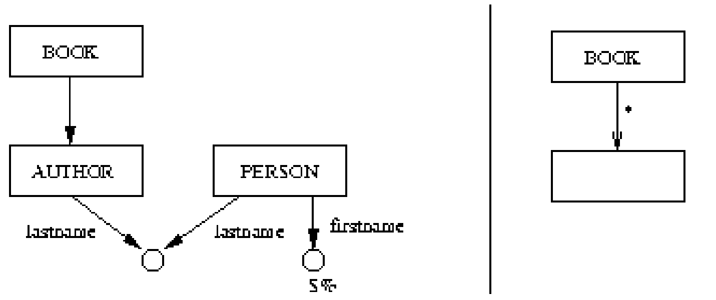
```
GOAL
results [
all co-authors [
name [ var Author ],
name [ var CoAuthor ]
]
]
FROM
bookdata {{
desc book {{
author {{ name {{ var Author }} }},
author {{ name {{ var CoAuthor }} }}
}}
END
```

This query profits from two features of Xcerpt: (1) Xcerpt's simulation unification is injective. This ensures that the two children of the book element in line 10 are different without requiring the query author to explicit state that the author and the co-author must be different. (2) Xcerpt's grouping is set based and uses unification for equality, i.e., two terms with same structure and values are considered equal even if they represent distinct elements in the input. Therefore the above program does not generate duplicates (as, e.g, the first XQuery solution for Query 9 in Section 7.3.1x).

A visual language, called *visXcerpt* [43, 42], has been conceived as a visual rendering of textual Xcerpt programs, making it possible to freely switch during programming between the visual and textual view, or rendering, of a program (cf. Figure 7.5 showing a textual and visual representation of an Xcerpt query).

Static type checking methods have been developed for Xcerpt [82, 372] that are based on seeing tree

Figure 7.6 Graph representation of an XML-GL query
([105], © Elsevier, Inc.)



grammars in their various disguises, e.g., DTD, XML Schema, RelaxNG, as definitions of abstract data type.

A declarative semantics for Xcerpt has been proposed in [90, 328]. A formal procedural semantics for Xcerpt has been proposed in [90] in the form of a a proof procedure. An implementation of this semantic in Haskell has been realized using Constraint Programming techniques [328]. The XQuery use case [107] has been worked out in Xcerpt (cf. [236] (in German) and [63]). Based on Xcerpt and extending it, a reactive language called XChange [86, 85] for updates and events on the Web is currently being developed.

Project page:
`http://www.xcerpt.org/`

Implementation:
available from the project page

Online demonstration:
`http://demo.xcerpt.org` and, using visXcerpt, `http://visxcerpt.xcerpt.org/`

7.3.2.8 XML-GL.

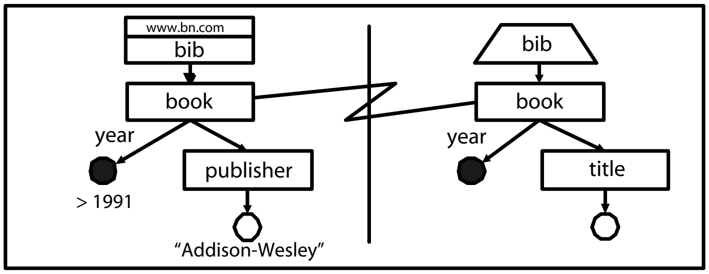
XML-GL [104, 105, 131] is a visual, rule-based query language for XML. Queries are specified as rules with a clear separation between query and construction. Queries are specified on the left-hand of a rule, construction on the right-hand. Figure 7.6 shows an XML-GL rule. Both sides of a rule are essentially (visual) patterns of the graph structure to be matched or constructed, but enriched with visual representations of a number of additional operators and functions (such as arithmetic operators, wildcards, predicates, negation, ordering, etc.). Connections between the two sides indicate where matched data occurs in the result.

Although XML-GL programs contain only a single rule, complex queries may contain multiple left-hand and right-hand sides for expressing set queries, such as unions, differences, cartesian product, and even heterogeneous unions. The original proposal of XML-GL does not allow recursive rules, but in [293] an extension of XML-GL in this direction is proposed.

Recently, a visual interface for XQuery, called XQBE [19, 67], based on XML-GL has been developed. Figure 7.7 shows the XQBE representation of the following XQuery expression (Query XMP-Q1 in [107]):

```
<bib>
{
```

Figure 7.7 XQBE Query
([67], © Elsevier, Inc.)



```
for $b in document("www.bn.com/bib.xml")/bib/book
where $b/publisher="Addison−Wesley" and $b/@year>1991
return <book year="{$b/@year}"> {$b/title} </book>
}
</bib>
```

Based on this visualization of XQuery expressions, an interactive editor for XQuery expressions is described in [67] (cf. Figure 7.8).

Project page:
XQBE: `http://dbgroup.elet.polimi.it/xquery/XQBE.html`

Implementation:
XQBE: available from the project page

Online demonstration:
none

7.3.2.9 X^2's visual interface.

X^2 [278] is a system for visual exploration and retrieval of XML databases. It provides an interactive environment for authoring visual queries, see Figure 7.9. The employed query language is rather restricted, but supports querying the order of elements and can be evaluated very efficiently (see [276]). Instead of constructing new data based on the results of a query, the system gathers all matched data in a novel data structure called "Complete Answer Aggregates" [277, 276] and allows the user to browse this structure, thereby exploring the data contained in the database. While browsing, the user can refine and reissue the query.

Project page:
`http://www.cis.uni-muenchen.de/people/Meuss/caa.html` and `http://www.cis.uni-muenchen.de/~weigel/Projekte/X2.html`

Implementation:
not publicly available

Online demonstration:
none

Figure 7.8 Screenshot of XQBE's query editor
([67], © Elsevier, Inc.)

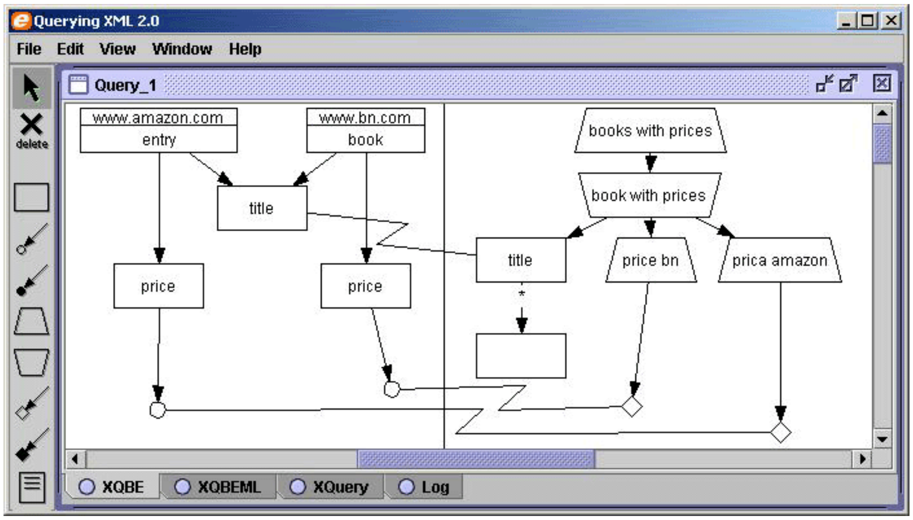
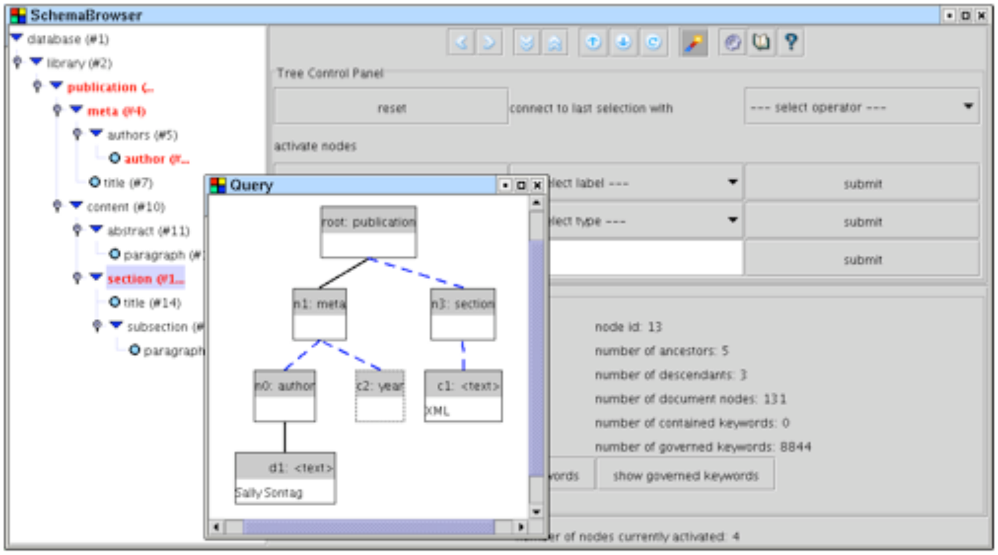


Figure 7.9 Screenshot of X²'s query editor
([278], © Springer-Verlag)



node id: 13
number of ancestors: 5
number of descendants: 3
number of document nodes: 131
number of contained keywords: 0
number of governed keywords: 8844

er of nodes currently activated: 4

7.4 RDF Query Languages

RDF Query Languages can be grouped into several families that differ in aspects like data model, expressivity, support for schema information, and kind of queries. As a "family", we consider languages that build upon each other, are heavily influenced by each other, or share a large part of their properties. In the following, we shall consider the six families *SPARQL*, *RQL*, *XPath-, XSLT-, and XQuery-based Languages*, *Metalog*, *Reactive Languages*, and *Deductive Languages*. In addition, we briefly introduce a number of additional languages that don't fall into one of the above-mentioned families.

7.4.1 The SPARQL Family

The SPARQL family consists of the four query languages *SquishQL*, *RDQL*, *SPARQL*, and *TriQL*. Common to all four languages in this family is that they "regard RDF as triple data without schema or ontology information unless explicitly included in the RDF source".

7.4.1.1 Basic RDF Access: SquishQL and RDQL.

The main objectives of SquishQL [282, 281] are ease-of-use and similarity to SQL. SquishQL relies on a query model for RDF influenced by [196]. SquishQL offers so-called "triple patterns" and conjunctions between triple patterns for specifying parts of RDF graphs to retrieved. *"This results in quite a weak pattern language but it does ensure that in a result all variables are bound."* [282]. SquishQL queries have the following form:

`SELECT` variables (identifies the variables whose bindings are returned)
`FROM` model URI
`WHERE` list of triple patterns
`AND` boolean expression (the filter to be applied to the result)
`USING` name `FOR` URI, …

In SquishQL, Query 20 can be expressed as follows:

```
SELECT ?essay, ?author, ?authorName
FROM http://example.org/books
WHERE (?essay, <rdf:type>, <books:Essay>),
(?essay, <books:author>, ?author),
(?author, <books:name>, ?authorName)
USING books FOR http://example.org/books#,
rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

In SquishQL, Query 21 can (almost) be expressed as follows:

```
SELECT ?property, ?propertyValue
FROM http://example.org/books
WHERE (?essay, <books:book-title>, "Bellum Civile")
(?essay, ?property, ?propertyValue),
USING books FOR http://example.org/books#
```

A property value can be a node with other properties, that an answer to Query 21 should return. Since SquishQL has no means to express recursion, such indirect properties cannot be returned by the above query if the schema of the data is unknown or recursive.

Other queries from Section 7.2.3 cannot be expressed in SquishQL.

In a SquishQL query, the `AND` clause serves to express constraints on variable values so as filter the bindings returned. The following query returns the URIs of persons that have authored a book with title "Bellum Civile".

```
SELECT ?person
FROM http://example.org/books
WHERE (?book, <books:author>, ?person)
(?book, <books:title>, ?title)
AND ?title = 'Bellum Civile'
```

An answer to an SquishQL query is a set of bindings for the variables occurring in the query. SquishQL does not support RDFS concepts.

Project page:

Inkling: http://swordfish.rdfweb.org/rdfquery/

Implementation:

Inkling [281]

RDQL, a "RDF Data Query Language", is an evolution of the SquishQL versions SquishQL [282], and Inkling [281] influenced by rdfDB [195]. RDQL has been recently submitted to the W3C for standardisation [332, 282, 334, 333]. RDQL queries have the same form as SquishQL queries. As with SquishQL, an answer to an RDQL query is a set of bindings for the variables occurring in the query. Like SquishQL, RDQL supports only selection and extraction queries.

RDQL is intentionally kept simple, operating only on the data level of RDF, with the goal to make RDQL amenable to standardisation as a "low-level RDF language". RDQL's authors see inferencing as a possible feature of an "RDF implementation", not of the query language RDQL: *"if a graph implementation provides inferencing to appear as 'virtual triples' (i.e. triples that appear in the graph but are not in the ground facts), then an RDQL query will include those triples as possible matches in triple patterns."* [332]. As a consequence, queries referring to RDFS relations such as type, set or class are cumbersome and/or complex.

The RDQLPlus (http://rdqlplus.sourceforge.net/) implementation of RDQL provides a language extension, called RIDIQL [373]. RIDIQL supports updates and a transparent use of the inference abilities of the Jena Toolkit [190].

Project pages:

http://www.hpl.hp.com/semweb/rdql.htm
RDFStore: http://rdfstore.sourceforge.net/

Implementations:

Jena Toolkit [334, 332, 333, 190], RAP (RDF API for PHP) [292], PHP XML Classes (http://phpxmlclasses.sourceforge.net/), RDFStore [319],
Rasqal (http://www.redland.opensource.ac.uk/rasqal/),
Sesame (http://www.openrdf.org/index.jsp),
RDQLPlus (http://rdqlplus.sourceforge.net/),
3store (http://sourceforge.net/projects/threestore/) [202].

Online demonstrations:

Sesame: http://www.openrdf.org/demo.jsp
RAP: http://www3.wiwiss.fu-berlin.de/rdfapi-php/test/custom_rdql_test.php
RDFStore: http://demo.asemantics.com/rdfstore/www2003/

SquishQL and RDQL queries cannot be composed. Negation can be used in filters, or AND clauses, as in the previous query, but not in WHERE clauses, i.e. triple patterns can only occur positively. Disjunctions and optional matching cannot be expressed. Although a variable in SquishQL and RDQL queries can be bound to blank nodes, there is no way to specify blank nodes in SquishQL's and RDQL's triple patterns. As a consequence, a query returning the blank nodes of a graph cannot be expressed in SquishQL and RDQL. SquishQL and RDQL have no form of recursion or iteration: By conjunction of triple patterns, one can express in SquishQL and RDQL only paths of a given length. Only selection and extraction queries can be expressed in SquishQL and RDQL, i.e., of the queries of Section 7.2.3, only Query 20 and (an approximation of) Query 21. Like SquishQL, RDQL does not support RDFS concepts, although at least one of its implementations, that given in the Jena Toolkit [190], supports the transitive closures of the RDFS relations `rdfs:subClassOf` and `rdfs:subPropertyOf`. No formal semantics has been defined for SquishQL or RDQL. The complexity of SquishQL and RDQL has not been investigated so far.

7.4.1.2 SPARQL.

SPARQL [316], a "Query Language for RDF" formerly called BrQL [315], has been developed by members of the W3C "RDF Data Access" Working Group. SPARQL is an extension of RDQL [332] designed according to requirements and use cases [124] and is still under development. SPARQL extends RDQL with facilities to:

- Extract RDF subgraphs.
- Construct, using `CONSTRUCT` clauses, one new RDF graph with data from the RDF graph queried. Like RDQL queries, the new graph can be specified with triple, or graph, patterns.
- Return, using `DESCRIBE` clauses, "descriptions" of the resources matching the query part. The exact meaning of "description" is not yet defined, cf. [351] for a proposal.
- Specify `OPTIONAL` triple or graph query patterns, i.e., data that should contribute to an answer if present in the data queried, but whose absence does not prevent to return an answer.
- Testing the absence, or non-existence, of tuples.

SPARQL queries have the following form:

`PREFIX`	Specification of a name for a URI (like RDQL's `USING`)
`SELECT`	Returns all or some of the variables bound in the `WHERE` clause.
`CONSTRUCT`	Returns a RDF graph with all or some of the variable bindings.
`DESCRIBE`	Returns a "description" of the resources found.
`ASK`	Returns whether a query pattern matches or not
`WHERE`	list, i.e., conjunction of query (triple or graph) patterns
`OPTIONAL`	list, i.e., conjunction of optional (triple or graph) patterns
`AND`	boolean expression (the filter to be applied to the result)

An extension of Query 20 returning the translators of a book, if there are some, can be expressed in SPARQL as follows:

```
PREFIX books: http://example.org/books#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
SELECT ?essay, ?author, ?authorName, ?translator
FROM http://example.org/books
WHERE (?essay books:author ?author),
(?author books:authorName ?authorName)
OPTIONAL (?essay books:translator ?translator)
```

Using the CONSTRUCT clause, restructuring and non-recursive inference queries can be expressed in SPARQL. Query 23 can be expressed in SPARQL as follows:

```
PREFIX books: http://example.org/books#
CONSTRUCT (?y books:authored ?x)
FROM http://example.org/books
WHERE (?x books:author ?y)
```

and Query 28 by

```
PREFIX books: http://example.org/books#
CONSTRUCT (?x books:co-author ?y)
FROM http://example.org/books
WHERE (?book books:author ?x)
(?book books:author ?y)
AND (?x neq ?y)
```

Project page:

http://www.w3.org/2001/sw/DataAccess/

Implementation:

none

Online demonstration:

none

7.4.1.3 TriQL.

TriQL extends RDQL by constructs supporting querying of named graphs [101], as introduced in TriG [55] by the authors of TriQL. Named graphs allow one to filter RDF statements after their sources or authors, like in the following query: *"Return the books with rating above a threshold of 5, using only information asserted by Marcus Tullius Cicero."* This can be expressed in TriQL as follows:

```
SELECT ?books
WHERE ?graph ( ?books books:rating ?rating )
(?graph swp:assertedBy ?warrant)
(?warrant swp:authority <http://people.net/cicero>)
USING books FOR http://example.org/books#,
swp FOR <http://www.w3.org/2004/03/trix/swp-1/>
```

Project page:

http://www.wiwiss.fu-berlin.de/suhl/bizer/TriQL/

Implementation:

none

Online demonstration:

none

7.4.2 The RQL Family

Under "RQL family", we group the three languages *RQL*, *SeRQL*, and *eRQL*. Common to these languages is that they support combining data and schema querying. Furthermore, the RDF data model they rely on slightly deviates from the standard data model for RDF and RDFS: (1) cycles in the subsumption hierarchy are forbidden, and (2) for each property, both a domain and a range must be defined. These restrictions ensure a clear separation of the three abstraction layers of RDF and RDFS: (1) data, i.e. description of resources such as persons, XML documents, etc., (2) schemas, i.e. classifications for such resources, and (3) meta-schemas specifying meta-classes such as `rdfs:Class`, the class of all classes, and `rdfs:Property` the class of all of properties. They make possible a flexible type system tailored to the specificities of RDF and RDFS.

7.4.2.1 RQL.

RQL, the "RDF Query Language", is developed at ICS-FORTH [119, 219, 218, 220, 221], and the base for the two other members of the RQL family, SeRQL and eRQL.

Basic schema queries. A salient feature of RQL is the use of the types from RDFS schemas. The query `subClassOf(books:Writing)` returns the sub-classes of the class `books:Writing`[9]. A similar query, using `subPropertyOf` instead of `subClassOf`, returns the the sub-properties of a property . The following query returns the domain (`$C1`) and range (`$C2`) of the property `author` defined at the URI named `book` (The prefix `$` indicates "class variable", i.e., a variable ranging on schema classes). It can be expressed in RQL in three different manners:

1. using class variables:

   ```
   SELECT $C1, $C2 FROM {$C1}books:author{$C2}
   USING NAMESPACE books = &http://example.org/books#
   ```

2. using a *type constraint*:

   ```
   SELECT C1, C2 FROM Class{C1}, Class{C2}, {;C1}books:author{;C2}
   USING NAMESPACE books = &http://example.org/books#
   ```

3. without class variables or type constraints:

   ```
   SELECT C1, C2 FROM subClassOf(domain(book:author)){C1},
   subClassOf(range(books:author)){C2}
   USING NAMESPACE books = &http://example.org/books#
   ```

The query `topclass(books:Historical_Essay)` returns the top of the subsumption hierarchy, i.e., `books:Writing`, cf. Figure 7.2. A similar query returns leaves of the subsumption hierarchy. The query `nca(books:Historical_Essay, books:Historical_Novel)` returns the nearest common ancestor of the classes of 'historical essays' and 'historical novels', i.e., the class `books:Essay` of 'essays'. RQL has "property variables" prefixed by `@` using which RDF properties can be queried (like classes using class variables). The following query, with property variables prefixed by `@`, similar to the formerly introduced class variables, returns the properties, together with their actual ranges, that can be assigned to resources classified as `books:Writing`:

```
SELECT @P, $V FROM {;books:Writing}@P{$V}
USING NAMESPACE books = &http://example.org/books#
```

Combining these facilities, Query 27 is expressible in RQL as follows:
`SELECT X, Y  FROM Class{X}, subClassOf(X){Y}`.

[9] Assuming: `USING NAMESPACE books = &http://example.org/books-rdfs#`

Data queries. With RQL, data can be retrieved by its types, by navigating to the appropriate position in the RDF graph. Restrictions can be expressed using filters. Classes, as well as properties, can be queried for their (direct and indirect[10]) extent. The query `books:Writing` returns the resources classified `books:Writing` or one of its sub-classes. This query can also be expressed as follows: `SELECT X FROM books:Writing{X}`. Prefixing the variable `X` in the previous queries, yields queries returning only resources directly classified as `books:Writing`, i.e., for which a statement (X, `rdf:type`, `books:Writing`) exists. The extent of a property can be similarly retrieved. The query `^books:author` returns the pairs of resources X, Y that stand in the `books:author` relation, i.e., for which a statement (X, `books:author`, Y) exists. RQL offers extended dot notation as used in OQL [103], for navigation in data and schema graphs. This is convenient for expressing Query 20:

```
SELECT X, Y, Z FROM {X;books:Essay}books:author{Y}.books:authorName{Z}
USING NAMESPACE books = &http://example.org/books#
```

The data selected by an RDF query can be restricted with a `WHERE` clause:

```
SELECT X, Y FROM {X;books:Essay}books:author.books:authorName{Y},
? {X}books:title{T}
WHERE T = "Bellum Civile"
USING NAMESPACE books = &http://example.org/books#
```

Mixed schema and data queries. With RQL, access to data and schema can be combined in all manners, e.g., the expression `X;books:Essay` restricts bindings for variable `X` to resources with type `books:Essay`. Types are often useful for filtering, but type information can also be interesting on their own, e.g., to return a "description" of a resource understood as its schema:

```
SELECT $C, ( SELECT @P, Y FROM {Z ; ^$D} ^@P {Y}
WHERE Z = X and $D = $C )
FROM ^$C {X}, {X}books:title{T} WHERE T = "Bellum Civile"
USING NAMESPACE books = &http://example.org/books#
```

This query returns the classes under which the resource with title "Bellum Civile" is directly classified; `^$C{X}` selects the values in the direct extent of any class.

Further features of RQL are not discussed here, e.g., support for containers, aggregation, and schema discovery. Although RQL has no concept of "view", extension RVL [254] of RQL gives a facility for specifying views. In RVL the inverse relation of `books:author` can be defined as a view as follows:

```
CREATE NAMESPACE mybooks = &http://example.org/books-rdfs-extension#
VIEW authored(Y, X) FROM {X}books:author{Y}
USING NAMESPACE books = &http://example.org/books#
```

RQL has been criticised for its large number of features and choice of syntactic constructs (like the prefixes `^` for calls and `@` for property variables), which resulted in the simplifications SeRQL and eRQL of RDF. RQL is far more expressive than most other RDF query languages, especially those of the SquishQL family. Most queries of Section 7.2.3, except those queries referring to the transitive closures of arbitrary relations, can be expressed in RQL: RDF supports only the transitive closures of `rdfs:subClassOf` and `rdfs:subPropertyOf`.

Query 20 is already given in RQL above. Query 21 cannot be expressed in RQL exactly, since RQL has no means to select "everything related to some resource". However, a modified version of this query, where a resource is described by its schema, is also given above. Reduction queries, e.g. Query 22, can often be concisely expressed in RQL, in particular if types are available:

[10] i.e. deduceable by inference.

```
SELECT S, @P, O
FROM (Resources minus (SELECT T FROM {B}books:translator{T})){S},
(Resources minus (SELECT T FROM {B}books:translator{T})){O},
{S}@P{O}
USING NAMESPACE books = &http://example.org/books#
```

An implementation of the restructuring Query 23 is given above in the extension RVL of RQL. RQL is convenient for expressing aggregation queries, e.g., Query 24:

```
max(SELECT Y
FROM {B;books:Writing}books:author.books:authorName{A},
{B}books:pubYear{Y}
WHERE A = "Julius Caesar")
```

Inference queries that do not need recursion, e.g., Query 28, can be expressed in RQL as follows:

```
SELECT A1, A2 FROM {Z}books:author{A1}, {Z}books:author{A2}
WHERE A1 != A2
USING NAMESPACE books = &http://example.org/books#
```

In RVL, an expression of Query 28 can actually create new statements as follows:

```
CREATE NAMESPACE mybooks = &http://example.org/books-rdfs-extension#
VIEW mybooks:co-author(A1, A2)
FROM {Z}books:author{A1}, {Z}books:author{A2} WHERE A1 != A2
USING NAMESPACE books = &http://example.org/books#
```

Both typing rules and a formal semantics for RQL have been specified [221]. No formal complexity study of RDF has been published yet. An implementation of RDF is given with the so-called "ICS-FORTH RDFSuite". RQL has influenced several later proposals for RDF query languages, e.g., BrQL and SPARQL, cf. Section 7.4.1.

Project page:
`http://139.91.183.30:9090/RDF/RQL/`

Implementation:
RDFSuite (`http://139.91.183.30:9090/RDF/index.html`)

Online demonstration:
`http://139.91.183.30:8999/RQLdemo/`

7.4.2.2 SeRQL.

SeRQL [95,78] is derived from RQL and differs from the latter as follows:

- SeRQL does not support RDF and RDFS types, except literal types.
- SeRQL modifies and extends RQL's path expressions. SeRQL compound path expressions instead use an "empty node", `{}`, for path concatenation. SeRQL provides a shorthand notation for retrieving several values of a property in a single path expression, simplifying, e.g., Query 28: In SeRQL, one can write `FROM {Book} <books:author> {X, Y}` instead of
  `FROM {Book} <books:author> {X}, {Book} <books:author> {Y}`.

  Furthermore, SeRQL supports optional path expressions (using square brackets), e.g.:

  ```
  SELECT  *  FROM {Book} <books:title> {Title};
       [ <books:translator> {Translator} [ <books:age> {Age} ] ].
  ```

- SeRQL provides a shorthand notation for expressing several properties of a resource in a FROM clause. The following SeRQL query returns the authors of books entitled "Bellum Civile" having a translator named "J.M. Carter" (note the ';' separating the different properties):

```
SELECT Author FROM {Book} <books:title> {"Bellum Gallicum"};
<books:translator>{}<books:translatorName>{"J.M. Carter"};
<books:author> {Author}
USING NAMESPACE books = <!http://example.org/books#>
```

- SeRQL eases querying a reified statement by enclosing the non-reified version of the statement in curly brackets.

SeRQL cannot express all queries of Section 7.2.3. Selection and extraction queries can be expressed in SeRQL (with the same limitation as with RQL, cf. above). In contrast to RQL, SeRQL has neither set operations, nor existential or universal quantification. As a consequence, Query 22 cannot be expressed in SeRQL. Thanks to the CONSTRUCT clause, SeRQL, like RQL, can express restructuring and simple inference queries, e.g., Query 23 can be expressed as:

```
CONSTRUCT {Author} <mybooks:authored> {Book}
FROM {Book} <books:author> {Author}
USING NAMESPACE books = <!http://example.org/books#>
mybooks = <!http://example.org/books-rdfs-extension#>
```

Aggregation queries cannot be expressed in SeQL (according to [95], adding aggregation to to SeRQL is planned). The transitive closure of `rdfs:subClassOf` is provided in SeRQL's implementation by means of the RDFS-aware storage of Sesame. However, neither the transitive closures of arbitrary relations nor general recursion can be expressed in SerQL.

Project page:
Sesame `http://www.openrdf.org/`

Implementation:
Implementation in Prolog[11]: `http://gollem.swi.psy.uva.nl/twiki/pl/bin/view/Library/SeRQL`

Online demonstrations:
`http://www.openrdf.org/demo.jsp`

7.4.2.3 eRQL.

eRQL [358] proposes a radical simplification of RQL based mostly on a keyword-based interface. It is the expressed goal of the authors of eRQL to provide with a "Google-like *query language but also with the capacity to profit of the additional information given by the RDF data*".[12] eRQL has only three query constructs:

One-word queries. Single words are valid eRQL queries, e.g., the query CAESAR returns all statements in which the string "CAESAR" occurs in any manner. Surprisingly, "phrase queries" like "Bellum civile" do not seem to be expressible in eRQL.

Neighbourhood queries. Neighbourhood queries are expressed by varying numbers of curly braces indicating the level of neighbourhood. They return not only the statements containing a word, as one-word queries, but also the statements related to ("in the neighbourhood of") a statement. For instance, the {{CAESAR}} returns the following statements (cf. Figure 7.2):

[11] Using the Semantic Web library of SWI Prolog `http://www.swi-prolog.org/`.

[12] `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`

```
_:1 books:author _:2. _:1 books:title "Bellum Civile".
_:1 books:authorName "Julius Caesar". _:1 books:translator _:4.
_:1 books:author _:3.
```

`{{{CAESAR}}}` extends the "neighbourhood" one step further, etc.

Conjunctive and disjunctive queries. Both, neighbourhood and one-word queries can be combined using the boolean operators `AND` and `OR`. No negation is provided, however.

Many queries of Section 7.2.3 cannot be expressed in eRQL. The extraction query Query 21 can be *approximated* in eRQL as: `{{"Bellum" AND "Civile"}}`. eRQL does not allow the selection of a neighbourhood of unknown size around a resource, e.g., for obtaining a "concise-bounded descriptions" [351]. Indeed, in contrast to the claims of eRQL's authors, this requires knowledge of the schema of the data queried. Nevertheless, the need for a language like eRQL is evident for exploiting RDF data with search engines.

Project page:

`http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`

Implementation(s):

eRQLEngine cf. project page

Online demonstration:

none

7.4.3 Query Languages inspired from XPath, XSLT or XQuery

This section is devoted to languages inspired from, or extending XML query languages. Some of them (viz. [324, 350, 366]) can be implemented with a few additional functions and/or by normalising the data before querying.

7.4.3.1 XQuery for RDF: The "Syntactic Web Approach".

[321, 324] propose to rely on the XML Query Language *XQuery* (cf. Section 7.3.1) for querying RDF data. The approach, called "Syntactic Web", consists of (1) a preliminary "normalisation" of the RDF data being queried essentially by (a) serialising RDF data in XML as collections of statements, and (b) grouping the statements by their subjects, and (2) defining in XQuery, functions conveying the semantics of RDFS, e.g., a function `rdf:instance-of-class` returning the (sequence of the) resources (represented by their `description` element) that are (direct or indirect) instances of a class:

```
define function rdf:instance-of-class($t as element(description)*,
$base-name as xs:string)
as element(description)*
{
$t[rdf:type = $base-name]
,
for $i in $t[rdfs:subClassOf = $base-name]
return rdf:instance-of-class($t, string($i/@rdf:about))
}
```

Using the function defined above, and assuming a convenient normalisation of the RDF data queried, Query 20 can be expressed as follows:

```
let $t := document("http://example.org/books")//description
for $essay in rdf:instance-of-class($t, "books:Essay"),
$author in $t[rdf:about = $essay/books:author]
return <result> {$essay, $author} </result>
```

The “Syntactic Web” approach also proposes a normalisation of Topic Maps and specific XQuery functions for querying Topic Maps data. This approach has several advantages. It makes it possible to return answers in any possible XML format and to query both, standard Web and Semantic Web data with the same query language, providing the uniformity advocated in [305]. [338] suggests a similar approach.

Project page:
none

Implementation(s):
not publicly available

Online demonstration:
none

7.4.3.2 XSLT for RDF: TreeHugger and RDF Twig.

Similar in spirit to the Syntactic Web Approach [321, 324], TreeHugger [350] proposes to rely on XSLT for querying and transforming RDF data. Due to limitations of XSTL 1.0, the normalisation of RDF data is not performed by an XSLT program, but by “extension functions”. The normalisation of RDF is based on the “striped syntax” [73], with properties represented both as elements and attributes (causing problems with multi-valued properties). Three extension functions are provided: (1) for loading an RDF document, (2) for loading an RDF document and handling the vocabulary of RDFS, and (3) for loading an RDF document and handling the vocabulary of both RDFS and OWL. XPath, upon which XSLT relies, is extended with a prefix `inv` for querying the inverse of an RDF property.

Query 20 can be expressed as follows in TreeHugger:

```
<results xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:books="http://example.org/books#"
xmlns:th="http://rootdev.net/net.rootdev.treehugger.TreeHugger"
xmlns:rdfs="http://www.w3.org/2000/01/rdf−schema#"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xsl:version="1.0">
<!-- Load RDF document -→
<xsl:variable name="doc"
select="th:documentRDFS(’http://example.org/books’)" />
<xsl:for-each select="$doc/books:Essay">
<xsl:for-each select="books:author/∗">
<result>
<xsl:value-of select="inv:books:author" />
<xsl:value-of select="." />
<authorName>
<xsl:value-of select="books:authorName/∗" />
</authorName>
</result>
</xsl:for-each>
</xsl:for-each>
</results>
```

Project page:
`http://rdfweb.org/people/damian/treehugger/`

Implementation(s):
Cf. project page

Online demonstration:
`http://swordfish.rdfweb.org/discovery/2003/09/treehugger/`

RDF Twig [366] is another extension of XSLT 1.0, with functions for querying RDF. It is based on "redundant" or "non-redundant" depth or breadth first traversals of the RDF graph, , i.e., traversals that repeat or do not repeat elements in the XML-based representation of RDF that are reachable from by various paths. Two query mechanisms are provided: A small set of logical operations on the RDF graph, and an interface to RDQL cf. Section 7.4.1.

Query 20 can be expressed as follows in RDF Twig:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
xmlns:rt="http://nwalsh.com/xslt/ext/com.nwalsh.xslt.saxon.RDFTwig"
xmlns:twig="http://nwalsh.com/xmlns/rdftwig#"
xmlns:books="http://example.org/books#"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#">
<xsl:template match="/">
<xsl:variable name="model"
select="rt:load('http://example.org/books')"/>
<!-- this is used as default model from now on-→
<xsl:variable name="pType"
select="rt:property('http://www.w3.org/1999/02/22−rdf−syntax−ns#',
'type')"/>
<xsl:variable name="essays"
select="rt:find($label, 'books:Essay')"/>
<xsl:variable name="tree"
select="rt:twig($essays)/twig:result"/>
<results>
<xsl:for-each select="rt:find($label, 'books:Essay')">
<result>
<xsl:value-of select="rt:twig(.)" />
<xsl:value-of select="rt:twig(.)/twig:result/books:author" />
</result>
</xsl:for-each>
</results>
</xsl:template>
```

Project page:
`http://rdftwig.sourceforge.net/`

Implementation:
Cf. project page

Online demonstration:
none

7.4.3.3 Versa.

Developed as part of the Python-based *4Suite* XML and RDF toolkit[13], Versa [290, 295, 291] is a query language for RDF inspired from, but significantly different to, XPath. Versa can be used in lieu of XPath in the XSLT version of 4Suite. Like the Syntactic Web Approach, TreeHugger, and RDF Twig, Versa is aligned with XML. Like XPath, Versa can be extended by externally defined functions. Versa's authors claim that Versa is easier to learn than RDF query languages inspired from SQL.

Versa has constructs for a *forward traversal* of one or more RDF properties, e.g., `all() - books:author -> *` selects those resources that are author of other resources. Instead of the wildcard $*$, string-based restrictions can be expressed. Using Versa's forward traversal operators, Query 20 can be expressed as follows:

[13] `http://4suite.org/`

```
distribute(type(books:Essay), ".",
"distribute(.−books:author→∗, ".", ".-books:authorName→∗)")
```

The function `distribute()` returns a list of lists containing the result of the second, third, … argument valuated starting from each of the resources selected by the first argument. As in XPath, `.` denotes the current node.

Versa has a *Forward filter* for selecting the subject of a statement, e.g., `type(books:Essay) |- books:title -> eq("Bellum Civile")` returns the essays entitled "Bellum Civile". Versa has also constructs for a *backward traversal* (but no backward filter), e.g., the essays entitled "Bellum Civile" can also be returned by `(books:Essay <- rdf:type - *) |- books:title -> eq("Bellum Gallicum")`. Versa's function `traverse` serves to traverse paths of arbitrary length, e.g., the following query returns all sub-classes of `books:Writing`:

```
traverse(books:Writing, rdf:subClassOf, vtrav:inverse, vtrav:transitive)
```

Similarly, Versa's function `filter` provides a general filter, e.g., all essays entitled "Bellum Gallicum" having a translator named "J. M. Carter" are returned by the following query:

```
filter(books:Essay <- rdf:type - ∗,
". − books:title →eq('Bellum Gallicum')",
". − books:translator →books:translatorName →eq('J. M. Carter')"
```

Selection and extraction queries can be easily implemented in Versa, although the selection of related items is not very convenient, as the above implementation of Query 20 demonstrates. In contrast to most RDF query languages, Versa allows the extraction of RDF subgraphs of arbitrary sizes, as required by Query 21. Reduction queries can be expressed in Versa, e.g., using negation or set difference. Query 22 can be implemented in Versa as follows:

```
difference(all(),
union(type(rdfs:Class),
union(type(rdfs:Property,
all() <- books:translator - ∗))
)
)
```

Restructuring, combination, and inference queries cannot be expressed in Versa, as the result of a Versa query is always a list (possibly a list of lists). However, Query 23 and 28 can be approximated in Versa as follows:

```
distribute(all(), ". − books:author →∗", ". − books:author →∗")
```

Answers to this query include "Julius Caesar" (as if he would be a co-author of himself !). This does not seem to be avoidable with Versa. Versa also provides several aggregation functions. Query 24 can be expressed as follows in Versa:

```
max(filter(all(),
". − books:author →books:authorName →eq('Julius Caesar')"
)
- books:year →∗)
```

Query 25 can be implemented in Versa using the function `length` as follows:

```
distribute(traverse(books:Writing, rdf:subClassOf,
vtrav:inverse,vtrav:transitive),
".",
"max(length((. <− rdf:type ∗) − books:author →∗))"
)
```

Neither a formal semantics, nor the language complexity have been investigated so far.

Project page:
http://uche.ogbuji.net/tech/rdf/versa/

Implementation(s):
available as part of 4Suite from http://4suite.org/

Online demonstration:
none

7.4.3.4 Path-Based Access to RDF: RDF Path, RPath, RxPath, RxSLT, and RxUpdate.

[302] sketches a language called RDF Path. RDF Path's syntax is similar to that of XPath. Node-tests for RDF data are added, e.g., `arc()` and `subj()`, and constructs of XPath not relevant for RDF are dropped. Functions and value tests are not considered in depth in this early draft. The fact that, in contrast to XML trees, RDF graphs do not have roots is not considered. As a consequence, finding a starting point for an RDF Path expression is an open issue.

Query 20 is not expressible, since related information cannot be selected. A variation of Query 21, *"Return the names of all authors of historical essays entitled 'Bellum Civile'."* can be expressed as follows:

```
*[rdf:type/books:Historical_Essay books:title/"Bellum Civile"]/
books:author/*/books:authorName
```

Project page:
http://infomesh.net/2003/rdfpath/

Implementation(s):
none

Online demonstration:
none

RPath [267] is another adaption of XPath to RDF, though focused on two RDF applications, CC/PP, a formalism for expressing device profiles, and UAProf, a formalism for expressing characteristics of (mobile) computers such as screen resolution and colour depth. RPath has location steps, *vertex-edge-tests* corresponding to node-tests in XPath, and predicates. RPath differences from XPath reflect the differences between the data models of XML and RDF, e.g., RPath's axes can follow a path along vertices (RDF predicates) and edges (RDF subjects and objects). As with RDF Path, the fact that RDF graphs are not rooted is not considered. Thus, it is not clear where an RPath expression should start from. This might not be too serious a problem, for the CC/PP and UAProf yield RDF graphs that are rooted two-level trees.

The variation of Query 21 considered above, *"Return the names of all authors of historical essays entitled 'Bellum Civile'."* can be expressed as follows:

```
/@vertex()[
rdf:type/@books:Historical_Essay and
books:title/@vertex()[equals('Bellum Civile')]
]/books:author/books:authorName
```

In contrast to most RDF query languages inspired from XPath, RPath does not require specifying paths where expressions match vertices, i.e., RDF classes, and edges (properties), alternate (like in striped RDF [73]). Thus, the previous query can also be expressed as follows:

```
outerVertex::vertex()[
outEdge::rdf:type/outVertex::books:Historical_Essay and
outEdge::books:title/outVertex::vertex()[equals('Bellum Civile')]
]/outEdge::books:author/outEdge::books:authorName
```

Project page:
none

Implementation(s):
prototype in Java, based on a CC/PP engine from Sun

Online demonstration:
none

RxPath is another adaption of XPath to RDF, defined within the project Rx4RDF[14], aiming at improving the accessibility of RDF for non-experts. In contrast to RDF Path and RPath, and similarly to TreeHugger and RDF Twig, RxPath is essentially *"a mapping between the RDF Abstract Syntax to the XPath Data Model"* [346]. This mapping is performed in four steps:

1. A top-level XML element is created for every RDF resource where the tag is the type of the resource,
2. "Each root element has a child element for each statement the resource is the subject of. The name of each child is [the] name of the property in the statement" [345],
3. "Each of these children have [a] child text node if the object of the statement is a literal or a child element if the object is a resource." [345], and
4. "Object elements have the same name and children as the equivalent root element for the resource, thus defining a potentially infinitely recursive tree." [345].

Since this mapping might lead to infinite trees, RxPath relies on a circularity-test for the evaluation of such axes ensuring that elements previously encountered are skipped (as a consequence, blank nodes have to be assigned a unique URI.) Furthermore, RxPath changes the semantics of the closure axes to only consider elements representing RDF properties in the original RDF model (this is easy as the mapping from RDF into an XML document discussed above uses a striped representation of RDF statements [73]). Finally, an expression such as `descendant::rdf:type` only matches an element representing an `rdf:type` property if all elements on the path to that property that represent any RDF property actually represent an `rdf:type` property. Thus, `descendant::rdf:type` is actually closer to the regular tree expression `(rdf:type._)*` than to the XPath expression `descendant::rdf:type`.

The variation of Query 21 considered above, *"Return the names of all authors of historical essays with the title 'Bellum Civile'."* can be expressed as follows (assuming the prefix `books` denotes `http://example.org/books-rdfs#`):

```
/books:Historical_Essay[books:title = 'Bellum Civile']/
books:author/*/books:authorName
```

Based on RxPath, two languages have been defined, RxSLT [347] and RxUpdate [348]. RxSLT is *"syntactically identical to XSLT 1.0"* [347], but uses RxPath instead of XPath 1.0. RxUpdate is syntactically very similar to XUpdate [247], but again uses RxPath instead of XPath to update RDF models. Note that RxSLT, like XSLT, is only capable of producing XML. Thus, new RDF data can only be created by using the XML serialisation of RDF.

[14] Phttp://rx4rdf.liminalzone.org/rx4rdf

Project page:
: `http://rx4rdf.liminalzone.org/rx4rdf`

Implementation(s):
: Cf. project page (prototype in Python)

Online demonstration:
: none

7.4.3.5 RDFT and the Query Language of Nexus: XSLT-Style RDF Query Languages.

RDFT [136] is a draft proposal closely related to XSLT 1.0. Like XSLT 1.0., RDFT uses templates that are matched recursively against the data structure. Since the structural recursion is performed against an RDF graph which can be cyclic, termination must be ensured. This issue has not yet been addressed. RDFT uses an adaption of XPath, called NodePath, for querying RDF graphs expressed in XML as "striped" [73]. Querying RDFS or OWL data has not yet been addressed.

RDFT only supports a subset of XSLT. A macro mechanism is introduced, as illustrated in lines 3–7 and 10 of the following implementation of Query 20 (for simplicity, only books and their authors are returned without considering the author's names):

```
<rt:stylesheet rt:version="1.0"
xmlns:rt="http://purl.org/vocab/2003/rdft/">
<rt:macro-set rt:prefix="rdf">
<rt:macro name="type"
value="resource(
'http://www.w3.org/1999/02/22−rdf−syntax−ns#type')/resource()"/>
</rt:macro-set>
<rt:root-template>
<rt:apply-templates
rt:select="/resource()[rdf:type =
resource("http://example.org/books#Essay")/>
</rt:root-template>
<!-- Template for the Essay
<rt:template pattern="resource()[rdf:type =
resource('http://example.org/books#Essay')" />
<xsl:value-of select="." />
<rt:apply-templates
rt:select="resource('http://example.org/books#author')/resource()/>
</rt:template>
<!−− Template for the author −→
<rt:template
pattern="resource('http://example.org/books#author')/resource()">
<xsl:value-of select="." />
</rt:template>
</rdft:stylesheet>
```

The [136] specification is not clear about the result of such a query: An XML tree or some form of an RDF graph? The description of `rt:element` seems to indicate the former, the description of `rt:value-of` the latter.

Project page:
: `http://www.semanticplanet.com/2003/08/rdft/spec`

Implementation(s):
: none

Online demonstration:
: none

[3] sketches another approach to querying RDF, and some form of XML, using an XSLT-like language. The basic idea is to translate RDF (expressed in XML) and also some non-RDF XML documents into a hierarchy of (attribute carrying) elements, based on the relations between the elements. The result of a query is some (hierarchical) view over this element tree. [3] does not address cyclic relations among elements but the language used seems to indicate that only proper hierarchies can be queried. RDF statements are mapped to nodes of an XML document as follows: Nodes represent RDF properties, an RDF statement (S, P, O) is represented by edges from all nodes representing some property with the value S to a node representing the property P with value O. A resource that never occurs as an object is assigned as value to a special property called `query:seed`. [3] seems to indicate that there can be only one such `query:seed` node, an assumption that does not hold for general RDF graphs. The query language provides a means for matching such property nodes based on the identifier (represented as URI or XML QName) of the property and the type (as determined by an `rdf:type` statement) of the value of the property.

Query 20 can be expressed as follows:

```
<query:plan>
<query:template match="query:seed" type="books:Essay">
<query:call name="query:insert" rename="book">
<query:call name="query:format" rename="title"
value="book:title" />
<query:call name="query:traverse" />
</query:call>
</query:template>
<query:template match="book:author">
<query:call name="query:insert" rename="author">
<query:call name="query:format" rename="name"
value="book:authorName" />
</query:call>
</query:template>
</query:plan>
```

An excerpt of the result of this query on the sample data from Figure 7.2 would be:

```
...
<book title="Bellum Civile">
<author name="Julius Caesar" />
<author name="Aulus Hirtius" />
</book>
...
```

Project page:
none

Implementation(s):
not publicly available, no report on any implementation

Online demonstration:
none

XsRQL: An XQuery-Style RDF Query Language. XsRQL [222], an XQuery-style RDF Query Language, is inspired from XQuery 1.0 [57], aiming at simplicity and flexibility. XsRQL departs from XQuery as follows: (1) The data model is adapted from RDF ([222] is rather vague on this point), (2) the path language considered is adapted to RDF and has only the axis `child`, (3) RDF properties are distinguished (from subjects and objects) by using `@`.[15]

Query 20 can be approximated in XsRQL as follows:

[15] In XPath, `@` indicate (flat) XML attributes. Since RDF properties are structured, in XsRQL a path expression may follow a `@` step.

```
declare prefix books: = <http://example.org/books#>;
declare prefix rdf: = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>;
for $essay in
datasource(<http://example.org/books>)//*[@rdf:type/books:Essay],
$author in $essay/@books:author/*
return
$essay, $author, $author/@books:authorName/*
```

XsRQL neither supports closure, nor a descendant-like axis, nor some other means of traversing an arbitrary-length path in the data structure. Therefore, it is not possible to also return resources classified by any sub-class of *books:Essay*.

Project page:
`http://www.fatdog.com/xsrql.html`

Implementation(s):
none

Online demonstration:
none

7.4.4 Metalog: Querying in Controlled English

Metalog [261, 262, 260] is a system for querying and reasoning with Semantic Web data. Its early proposal has led to the claim that *"Metalog has been the first semantic web system to be designed, introducing reasoning within the Semantic Web infrastructure by adding the query/logical layer on top of RDF"* cf. `http://www.w3.org/RDF/Metalog/`. Metalog notably differs from other RDF query languages for two reasons: (1) Metalog combines querying with *reasoning*, and (2) the language syntax is a controlled natural language (English), i.e., a non-ambiguous language reminding of natural language.

Query 20 can be expressed in Metalog as follows:

```
comment: some definitions of variables (or representations)
ESSAY represents the term "Essay"
from the ontology "http://example.org/books#".
AUTHORED-BY represents the verb "author"
from the ontology "http://example.org/books#".
IS represents the verb "rdf:type"
from RDF "http://www.w3.org/1999/02/22−rdf−syntax−ns#".
BELLUM_CIVILE represents the book "Bellum_Civile"
from the collection of books "http://example.org/books#".
comment: RDF triples written as Metalog statements.
BELLUM_CIVILE IS an ESSAY.
BELLUM_CIVILE is AUTHORED-BY "Julius Caesar".
BELLUM_CIVILE is AUTHORED-BY "Aulus Hirtius".
comment: a Metalog query
do you know SOMETHING that IS an ESSAY and that is AUTHORED-BY SOMEONE?
```

Project page:
`http://www.w3.org/RDF/Metalog/`

Implementation(s):
Cf. project page

Online demonstration:
none

Table 7.1 Answer to Query 20

?title	?translator	*Proof*
“Bellum Civile”	“J. M. Carter”	_:1 rdf:type <http://exam...ks-rdfs#Essay>. _:1 books:author _:2. _:2 books:authorName ‘‘Julius Caesar’’. _:1 books:title ‘‘Bellum Civile’’. _:1 books:translator ‘‘J. M. Carter’’.

7.4.5 Query Languages with Reactive Rules.

7.4.5.1 Algae.

Algae[16] is an RDF query language developed as part of the W3C Annotea project (`http://www.w3.org/2001/Annotea/`) aiming at enhancing Web pages with semantic annotations, expressed in RDF and collected from ‘annotation servers’, as Web pages are browsed. Algae is based on two concepts: (1) “Actions” are the directives `ask`, `assert`, and `fwrule` that determine whether an expression is used to query the RDF data, insert data into the graph, or to specify ECA-like rules. (2) Answers to Algae queries are bindings for query variables as well as triples from the RDF graph as “proofs” of the answer. Algae queries can be composed. Syntactically, Algae is based on the RDF syntax N-triples [186]. Algae extends the N-triple syntax with the above mentioned “actions” and with so-called “constraints”, written between curly brackets, that specify further arithmetic or string comparisons to be fulfilled by the data retrieved.

Query 20 can be expressed as follows:

```
ns rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ns books = <http://example.org/books#>
read <http://example.org/books> ()
ask ( ?essay rdf:type <http://example.org/books#Essay> .
?essay books:author ?author .
?author books:authorName ?authorName )
collect( ?essay, ?author, ?authorName )
```

This query becomes more interesting if we are not only interested in the titles of essays written by “Julius Caesar” but also want the translators of such books returned, if there are any:

```
ns rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ns books = <http://example.org/books#>
read <http://example.org/books> ()
ask ( ?essay rdf:type <http://example.org/books#Essay> .
?essay books:author ?author .
?author books:authorName ‘‘Julius Caesar’’ .
?essay books:title ?title .
~?essay books:translator ?translator .
)
collect( ?title, ?translatorName )
```

Note ~ used to declare ‘translator’ an optional. This query returns the answer given in Table 7.1.

Query 21 and Query 23 cannot be expressed in Algae due to the lack of closure, recursion, and negation. Queries 24 and 25 cannot be expressed in Algae due to the lack of aggregation operators. All other queries can be expressed in Algae, most of them requiring ‘extended action directives’ [311].

No formal semantics has been published for Algae.

Project page:

`http://www.w3.org/2004/05/06-Algae/` and for the Annotea project `http://www.w3.org/2001/Annotea/`

[16] Also called “Algae2”. This survey follows [312] and retains the name “Algae”.

Implementation(s):

W3C Annotation Server `http://annotest.w3.org/annotations`

Online demonstration:

Query interface to the W3C Annotation Server using Algae as query language: `http://annotest.w3.org/annotations?explain=false`

7.4.5.2 iTQL.

iTQL, a query and update language, has been defined for Kowari Metastore, an open source database for the storage of RDF data. iTQL offers commands for querying, `select`, updating, `delete` and `insert`, and transaction management, `commit` and `rollback`. The syntax of iTQL is reminiscent of SQL, and therefore also of RDQL. The querying capabilities of iTQL are limited like those of RDQL: iTQL supports only simple selections. iTQL allows nested queries.

Query 20 can be expressed as follows in iTQL:

```
alias <http://example.org/books#> as books;
alias <http://www.w3.org/2000/01/rdf-schema#> as rdfs;
alias <http://www.w3.org/1999/02/22-rdf-syntax-ns#> as rdf;
select $essay, $author, $authorName
where $essay <books:author> $author
and $author <books:authorName> $authorName
and $essay <rdf:type> $type
and (trans($type <rdfs:subClassOf> <books:Essay>)
or $type <tks:is> <books:Essay>)
```

iTQL's function `trans` computes the transitive closure of a relation, in the example of `rdfs:subClassOf`. Paths of arbitrary length in an RDF graph can be traversed using iTQL's function `walk`. Like SQL, iTQL allows sorted answers and accessing answers in a paged mode using `limit` and `offset`.

Project page:

`http://www.kowari.org`

Implementations:

Kowari Metastor
Tucana Knowledge Server

Online demonstration:

none

7.4.5.3 WQL.

WQL, Wilbur Query Language, is the name given in [244] to query primitives of Ivanhoe [243], a frame-based API inspired from [212, 240] for the Nokia Wilbur Toolkit [245], a collection of APIs for XML, RDF, and DAML written in CLOS, Common Lisp Object System [241].

In WQL, like in Ivanhoe, a RDF or DAML resource is represented as a frame with a slot for each property. The (possibly multiple) values of a slot correspond to objects of RDF statements, with the resources represented by the frame as subjects. Three WQL variants are discussed and compared in [200]:

- a basic query language, WQL proper, with constructs `value` and `all-values` for a path-based selection of one or all resources, and `relatedp` for testing resource relations.

- an embedding, called WQL+CL, of the above-mentioned basic language in Common Lisp.[17]
- WQL+CL+inference, an extension of WQL+CL, with a data store providing inferencing based upon the “transparent” (or “hidden”) inference extensions described in [242].

In the following, WQL proper and, where appropriate, the “transparent inferencing” of WQL+CL+inference are considered. WQL+CL is not considered, for it is more akin to a programming language than a query language.

The following query returns the labels of all classes the book identified by http://example.org/books#Bellum_Civile belongs to:

```
(setf *db* (make-instance 'db))
(load-db (make-url "http://example.org/books")
:locator "http://example.org/books")
(add-namespace "books" "http://example.org/books#")
(all-values !"http://example.org/books#Bellum_Civile"
'(:seq !rdf:type (:seq (:rep* !rdfs:subClassOf) !rdfs:label)))
```

Note :seq constructing a sequence of slots, i.e., RDF relations, to be traversed by the query and :rep* traversing the transitive closure of a slot/relation. all-values returns all resources, (represented as frames, reachable on the specified path from the source frame, i.e., the frame with identifier http://example.org/books#Bellum_Civile.

Project page:
Wilbur Toolkit: http://wilbur-rdf.sourceforge.net/

Implementation:
Cf. project page

Online demonstration:
none

7.4.6 Deductive Query Languages

7.4.6.1 N3QL.

N3QL, sketched in [48], is derived from the rule fragment of Notation 3 [49] (shorthand N3), a syntax for and extension of RDF with variables, rules, and quoting for easy expression of statements about statements. N3QL differs from the rule fragment of N3 in that its syntax has “query language style” clauses such as select and where.

An N3QL query is an N3 expression and all N3QL reserved words are the RDF properties of an RDF (usually, but not necessarily) blank node representing the query.

Query 20 can be expressed as follows in N3QL:

```
@prefix books: <http://example.org/books#>.
@prefix n3ql: <http://www.w3.org/2004/ql#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
[] n3ql:select { n3ql:result n3ql:is (?book ?author ?authorName) };
n3ql:where { ?book rdf:type books:Essay;
?book books:author ?author;
?author books:authorName ?authorName }.
```

[17] It is unclear whether WQL+CL restricts Common Lisp.

The answer to this query is the RDF graph specified in the `n3ql:select` clause, a set of RDF collections (indicated by the collection constructor `()`) of bindings for the three variables.

[48] seems to indicate that a N3QL query is equivalent to a N3 rule, the `where` part of the N3QL query being the rule's premise, and the `select` part, the rule's consequence. However, whereas N3 rules can express transitive closures, this is not the case of N3QL queries. The following N3 rule specifies the transitive closure of a RDF property:

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
{?x rdfs:subClassOf ?z; ?z rdfs:subClassOf ?y}
=> {?x rdfs:subClassOf ?y}
```

Note that the description of N3QL does not clearly specify which of the syntactic constructs of N3 can be used in N3QL. [48] states that N3QL is a restricted form of N3 where formulae cannot be nested and literals cannot be subjects of statements. The N3 syntax for anonymous nodes, for navigating in the RDF graph using path expressions, and for quantified variables gives rise to concise expressions of of queries such as *"Return the books written by the author named 'Julius Caesar'."*:

```
@prefix books: <http://example.org/books#>.
@prefix n3ql: <http://www.w3.org/2004/ql#>.
[] n3ql:select { n3ql:result n3ql:is (?book) };
n3ql:where { ?book!books:author!books:authorName ‘‘Julius Caesar’’ }.
```

Project page:
`http://www.w3.org/DesignIssues/N3QL.html`

Implementations:
CWM `http://www.w3.org/2000/10/swap/doc/cwm.html`
EulerSharp `http://eulersharp.sourceforge.net/2003/03swap/`

Online demonstration:
none

7.4.6.2 R-DEVICE.

R-DEVICE [32] is a *"deductive object-oriented knowledge-base system for querying and reasoning about RDF metadata."*[18] It is a reimplementation of the X-DEVICE language [31] in the C Language Integrated Production System, or CLIPS, cf. `http://www.ghg.net/clips/CLIPS.html`, using the CLIPS Object-Oriented Language, COOL. RDF triples are mapped to objects as follows:

- RDF resources are represented as objects, the types of which are the resource's RDF types, i.e., the values of the `rdf:type` properties. For resources that are classified in multiple classes, a 'dummy class' is introduced which represents a common subclass of all the classes the resource is classified in.

- RDF properties are realized as multi-slots, i.e., slots with multiple values, in the class which is the domain of the property. If no domain is given, i.e., if the property can be applied to any resources, a slot is added to the class representing `rdfs:Resource`, the top of the RDF resource hierarchy.

Assertions generated, e.g., through rules, can require dynamic class and/or object re-definitions.

Query 20 can be expressed as follows:

[18] `http://lpis.csd.auth.gr/systems/r-device.html`

```
(deductiverule q1
?book <- (? (rdf:type books:Essay) (books:author ?author))
?author <- (? (books:authorName ?authorName))
=>
(result (book ?book) (author ?author) (authorName ?authorName))
)
```

Note the production-rule like syntax of R-DEVICE.

R-DEVICE provides constructs for traversing arbitrary length paths of slots and objects, properties and resources, both with and without restriction on the type of slot that may be traversed. This allows one to implement both Query 21 and Query 27. Query 21 can be expressed as follows:

```
(deductive rule q2
?book <- (? (rdf:type books:Essay) (books:title ‘‘Bellum Civile’’)
(($?p) ?related)
=>
(result (book ?book) (related ?related))
)
```

Project page:
`http://lpis.csd.auth.gr/systems/r-device.html`

Implementation:
Cf. project page

Online demonstration:
none

7.4.6.3 TRIPLE

TRIPLE [341, 342, 203] is a rule-based query, inference, and transformation language for RDF. TRIPLE is based upon ideas published in [138]. TRIPLE's syntax is close to F-Logic [230]. F-Logic is convenient for querying semi-structured data, e.g., XML and RDF, as it facilitates describing schema-less or irregular data [250]. Other approaches to querying XML and/or RDF are XPathLog and the ontology management platform Ontobroker[19]. TRIPLE has been designed to address two weaknesses of previous approaches to querying RDF: (1) Predefined constructs expressing RDFS' semantics that restrain a query language's extensibility, and (2) lack of formal semantics.

Instead of predefined RDFS-related language constructs, TRIPLE offers Horn logic rules (in F-Logic syntax) [230]. Using TRIPLE rules, one can implement features of, e.g., RDFS. Where Horn logic is not sufficient, as is the case of OWL, TRIPLE is designed to be extended by external modules implementing, e.g., an OWL reasoner. Thanks to its foundations in Horn logic, TRIPLE can inherit much of Logic Programming's formal semantics. Referring to, e.g., a representation of UML in RDF [238, 239], the authors of TRIPLE claim in [342] that TRIPLE is well-suited to query non-RDF meta-data. This can be questioned, especially if, in spite of [178], one considers the rather awkward mappings of Topic Maps into RDF proposed so far.

TRIPLE differs from Horn logic and Logic Programming as follows [342]:

- TRIPLE supports resources identified by URIs.
- RDF statements are represented in TRIPLE by slots, allowing the grouping and nesting of statements; like in F-Logic, Path expressions inspired from [169] can be used for traversing several properties.

[19] `http://www.ontoprise.de/products/ontobroker`

- TRIPLE provides concise support for reified statements. Reified statements are expressed in TRIPLE enclosed angle brackets, e.g.:
  `Julius_Caesar[believes-><Junius_Brutus[friend-of -> Julius_Caesar]>]`
- TRIPLE has a notion of module allowing specification of the 'model' in which a statement, or an atom, is true. 'Models' are identified by URIs that can prefix statement or atom using `@`.
- TRIPLE requires an explicit quantification of all variables.

Query 20 can be approximated as follows:

```
rdf := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
books := 'http://example.org/books#'.
booksModel := 'http://example.org/books'.
FORALL B, A, AN result(B, A, AN) <-
B[rdf:type →books:Essay;
books:author →A[books:authorName →AN]]@booksModel.
```

This query selects only resources directly classified as `books:Essay`. Query 20 is properly expressed below.

TRIPLE's rules give rise to specify properties of RDF. [342] gives the following implementation of a part of RDFS's semantics:

```
rdf := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
rdfs := 'http://www.w3.org/2000/01/rdf-schema#'.
type := rdf:type.
subPropertyOf := rdfs:subPropertyOf.
subClassOf := rdfs:subClassOf.
FORALL Mdl @rdfschema(Mdl) {
transitive(subPropertyOf).
transitive(subClassOf).
FORALL O,P,V O[P→V] <-
O[P→V]@Mdl.
FORALL O,P,V O[P→V] <-
EXISTS S S[subPropertyOf→P] AND O[S→V].
FORALL O,P,V O[P→V] <-
transitive(P) AND EXISTS W (O[P→W] AND W[P→V]).
FORALL O,T O[type→T] <-
EXISTS S (S[subClassOf→T] AND O[type→S]).
}
```

Inference from range and domain restrictions of properties are not implemented by the rule given above. This is not limitation of TRIPLE, though, for the following rules provides them:

```
FORALL S,T S[type-$>$T] <-
EXISTS P, O (S[P-$>$O] AND P[rdfs:domain-$>$T]).
FORALL O,T O[type→T] <-
EXISTS P, S (S[P-$>$O] AND P[rdfs:range-$>$T]).
```

With the rules given above, the approximation of Query 20 given above only needs to be modified so as to express the 'model' it is evaluated against: instead of `@booksModel`, `@rdfschema(booksModel)` should be used, i.e., the original 'model' should be extended with the above-mentioned implementing RDFS' semantics. Most queries of Section 7.2.3 can be expressed in TRIPLE. Aggregation queries cannot be expressed in TRIPLE, for the language does not support aggregation.

[342] specifies an RDF, and therefore XML, syntax for a fragment of TRIPLE. By relying on translations to RDF, one can query data in different formalisms with TRIPLE, e.g., RDF, Topic Maps, and UML. This, however, might lead to rather awkward queries. Some aspects of RDF, viz. containers, collections, and anonymous nodes, are not supported by TRIPLE. The complexity of TRIPLE has not been investigated so far.

Project page:

`http://triple.semanticweb.org/`

Implementation:

Cf. project page

Online demonstration:

Cf. project page
`http://ontoagents.stanford.edu:8080/triple/`[20]

7.4.6.4 Xcerpt.

Xcerpt [88,87,329,41], cf. `http://xcerpt.org`, is a language for querying both data on the "standard Web" (e.g., XML and HTML data) and data on the Semantic Web (e.g., RDF, Topic Maps, etc. data). Using Xcerpt for querying XML data is addressed in Section 7.3.2.7. This Section is devoted to applying Xcerpt to querying RDF data.

Three features of Xcerpt are particularly convenient for querying RDF data. (1) Xcerpt's pattern-based incomplete queries are convenient for collecting related resources in the neighbourhood of some given resources and to express traversals of RDF graphs of indefinite lengths. (2) Xcerpt chaining of (possibly recursive rules) is convenient for expressing RDFS's semantics, e.g., the transitive closure of the `subClassOf` relation, as well as all kinds of graph traversals. (3) Xcerpt's optional construct is convenient for collecting properties of resources.

All nine queries from Section 7.2.3 can be expressed in Xcerpt's both on the XML serialization (cf. Section 7.3.2) and on the RDF serialization of the sample data from Section 7.2.2. The following Xcerpt programs show solutions for the queries against the RDF serialization.

[62] proposes two views on RDF data: as in most other RDF query languages as plain triples with explicit joins for structure traversal and as a proper graph.

On the plain triple view, Query 1 can be expressed in Xcerpt as follows:

```
DECLARE ns-prefix rdf = "http://www.w3.org/1999/02/22−rdf−syntax−ns#"
DECLARE ns-prefix books = "http://example.org/books#"
GOAL
result [
all essay [
id [ var Essay ],
all author [
id [ var Author ],
all name [ var AuthorName ]
]
]
]
FROM
and{
RDFS-TRIPLE [
var Essay:uri{}, "rdf:type":uri{}, "books:Essay":uri{}
],
RDF-TRIPLE [
var Essay:uri{}, "books:author":uri{}, var Author:uri{}
],
RDF-TRIPLE [
var Author:uri{}, "books:authorName":uri{}, var AuthorName
]
}
END
```

[20] Not functioning at the time of writing.

Using the prefixes declared in line 1 and 2, the query pattern (between FROM and END) is a conjunction of tree queries against the RDF triples represented in the predicate RDF-TRIPLE. Notice that the first conjunct actually uses RDFS-TRIPLE. This view of the RDF data contains all basic triples plus the ones entailed by the RDFS semantics [204] (cf. [62] for a detailed description). Using RDFS-TRIPLE instead of RDF-TRIPLE ensures that also resources actually classified in a sub-class of books:Essay are returned.

Xcerpt's approach to RDF querying shares with [321] and a few other approaches in Section 4.3 the ability to construct arbitrary XML as in this rule.

On Xcerpt's graph view of RDF, the same query can be expressed as follows:

```
DECLARE ns-prefix rdf = "http://www.w3.org/1999/02/22−rdf−syntax−ns#"
DECLARE ns-prefix books = "http://example.org/books#"
GOAL
result [
all essay [
id [ var Essay ],
all author [
id [ var Author ],
all name [ var AuthorName ]
]
]
]
FROM
RDFS-GRAPH {{
var Essay:uri {{
rdf:type {{ "books:Essay":uri {{ }} }},
books:author {{
var Author:uri {{
books:name {{ var AuthorName }}
}}
}}
}}
}}
END
```

The RDF graph view is represented in the RDF-GRAPH predicate. Here, the RDFS-GRAPH view is used that extends RDF-GRAPH as RDFS-TRIPLE extends
RDF-TRIPLE. Triples are represented similar to striped RDF/XML: each resource is a direct child element in RDF-GRAPH with a sub-element for each statement with that resource as object. The sub-element is labeled with the URI of the predicate and contains the object of the statement. As Xcerpt's data model is a rooted *graph* this can be represented without duplication of resources.

In contrast to the previous query no conjunction is used but rather a nested pattern that naturally reflects the structure of the RDF graph with the exception that labeled edges are represented as nodes with edges to the elements representing their source and sink.

To illustrate this graph view, consider the following rule showing how to generate the graph view from the triple view introduced above:

```
CONSTRUCT
RDF-GRAPH {
all var Subject @ var Subject:var SubjectType {
all optional var Predicate {
^var Object
},
all optional var Predicate {
var Literal
}
} }
FROM
or{
RDF-TRIPLE[
var Subject:var SubjectType{},
var Predicate:uri{},
optional var Literal as literal{{}},
optional var Object:/uri|blank/{{}}
],
RDF-TRIPLE[
/.*/:/.*/{{}},
/.*/:/.*/{{}},
var Subject:var SubjectType{{}}
] }
END
```

Notice the use of the `optional` keyword in lines 16 and 17. This indicates that the contained part of the pattern does not have to occur in the data, but if it does occur the contained variables are bound appropriately. `Optional` allows queries with alternatives to be expressed very concisely and is therefore crucial for RDF where all properties are optional by default.

In lines 3 and 5 the construction of a graph is shown: by using the operators `@` and `^` a (possibly cyclic) link can be constructed.

Xcerpt rules are convenient for making the language "RDF serialisation transparent". For each RDF serialisation, a set of rules expresses a translation from or into that serialisation. However, the rules for parsing RDF/XML [38], the official XML serialisation, are very complex and lengthy due to the high degree of flexibility RDF/XML allows. They can be found in [62], similar functions for parsing RDF/XML in XQuery are described in [324]. The following rules parse RDF data serialised in the RXR (Regular XML RDF) format [21], a far simpler and more regular RDF serialisation.

The following rule extracts all triples from an RXR document. Since different types (such as URI, blank node, or literal) of subjects and objects of RDF triples are represented differently in RXR, the conversion of the RXR representation into the plain triples is performed in separate rules, see [62].

```
DECLARE ns-prefix rxr = "http://ilrt.org/discovery/2004/03/rxr/"
CONSTRUCT
RDF-TRIPLE[
var Subject, var Predicate:uri{}, var Object
]
FROM
and[
rxr:graph {{
rxr:triple {
var S →rxr:subject{{}},
rxr:predicate{ attributes{ rxr:uri{ var Predicate } } },
var O →rxr:object{{}}
}
}},
RXR-RDFNODE[ var S, var Subject ],
RXR-RDFNODE[ var O, var Object ]
]
END
```

Querying RDF data with Xcerpt is the subject of ongoing investigation [62].

7.4.7 Other RDF Query Languages

RDF-QBE [320] is inspired from QBE [378,379], the database query language that introduced the celebrated "Query by Example" paradigm. An RDF graph, expressed in the syntax of Notation 3 [49]), is used to describe query patterns, variables are expressed as blank nodes that, according to [234] doe not have explicit identifiers. The representation of variables as blank nodes leads to a major restriction of RDF-QBE : Query patterns can only be tree-shaped.[21] RDF-QBE is especially convenient for expressing selection and extraction queries. However, the expressive power of RDF-QBE is limited: Not all queries of Section 7.2.3 can be expressed.

[21] [320] (wrongly) suggests that this restriction reduces query-answering to tree matching because the data queried is not necessarily tree-shaped.

Project page:
none

Implementation:
described in [320] but not publicly available

Online demonstration:
none

7.4.7.1 RDFQL.

RDFQL is the query language of RDF Gateway [214], a platform for developing and deploying Semantic Web applications combining a "native" RDF database engine, a Web server, and a server-side scripting language. The RDF database engine allows for the integration of standard and Semantic Web using so-called "virtual tables" and inference rules for deductive reasoning (so far, libraries for OWL and RDFS are provided). RDF Gateway supports several serialisations of RDF, viz. RDF/XML, N3, and NTriples. Although similar to RDQL, cf. Section 7.4.1, RDFQL differs from RDQL as follows: (1) RDFQL includes database commands for transaction management, e.g., commit and rollback, (2) RDFQL includes SQL-like update commands, (3) RDFQL allows accessing data from disk-based, in-memory, or external[22] "data sources", and (4) RDFQL's command `INFER` allows specification of deduction rules to be used when querying.

With RDFQL's rules, the semantics of RDFS can be expressed as follows:

```
RULEBASE rdfs
{
INFER {[rdf:type] ?a [rdf:Property]} from {?a ?x ?y};
INFER {[rdf:type] ?x ?z} from {[rdfs:domain] ?a ?z} and {?a ?x ?y};
INFER {[rdf:type] ?u ?z} from {[rdfs:range] ?a ?z}
and {?a ?x ?u} and uri(?u)=?u;
INFER {[rdf:type] ?x [rdfs:Resource]} from {?a ?x ?y};
INFER {[rdf:type] ?u [rdfs:Resource]} from {?a ?x ?u} and uri(?u)=?u
INFER {[rdfs:subPropertyOf] ?a ?c}
from {[rdfs:subPropertyOf] ?a ?b} and {[rdfs:subPropertyOf] ?b ?c}
INFER {?b ?x ?y} from {[rdfs:subPropertyOf] ?a ?b}
and {?a ?x ?y}
INFER {[rdfs:subClassOf] ?x [rdfs:Resource]}
from {[rdf:type] ?x [rdfs:Class]}
INFER {[rdfs:subClassOf] ?x ?z} from {[rdfs:subClassOf] ?x ?y}
and {[rdfs:subClassOf] ?y ?z}
INFER {[rdf:type] ?a ?y} from {[rdfs:subClassOf] ?x ?y}
and {[rdf:type] ?a ?x}
}
```

`{?P ?S ?O}` denotes in RDFQL an RDF statement with subject *S*, property *P*, and object *O*, i.e., RDFQL uses a prefix notation for RDF statements. `uri(?u)=?u` serves to detect whether the object of an RDF statement is a resource (in which case it has an URI and this URI is equal to the "value" of the resource itself) or a literal.

Query 20 can be implemented as follows:

```
session.namespaces["books"] = "http://example.org/books#";
var booksdata = new DataSource("http://example.org/books");
SELECT ?essay, ?author, ?authorName USING booksdata WHERE
{[rdf:type] ?essay [books:Essay]}
and {[books:author] ?essay ?author}
and {[books:authorName] ?author ?authorName}
ORDER BY ?authorName DESC;
```

[22] I.e., identified, e.g., by an URI.

Project page:
`http://www.intellidimension.com/`

Implementations:
RDF Gateway
Cf. project page for a limited, non-commercial use

Online demonstration:
none[23]

7.5 Topic Maps Query Languages

7.5.1 tolog: Logic Programming for Topic Maps

tolog [174,175,172,173] is the query language of the Ontopia Knowledge Suite[24]. tolog has also been selected in April 2004 as an initial straw-man for the ISO Topic Maps Query Language. tolog is inspired from Logic Programming and has SQL-style constructs. tolog provides a means for identifying a topic by its (internal) identifier and its subject indicator, e.g., the topic (type) "Novel" of the sample data can be accessed either by its identifier `Novel`, or its subject indicator `i"http://example.org/books#Novel"`.[25] URI prefixes can be used, e.g., `using books for i"http://example.org/books#"` gives rise to the short form `books:Novel` for the above-mentioned subject indicator. Note that tolog URI prefixes contain indicators and therefore differ from XML namespaces. In tolog, all occurrences of variables must be prefixed with `$`.

The original version of tolog [174]) has two kinds of Prolog-like "predicates", "built-in" and "dynamic association predicates". tolog has a "dynamic association predicate" for querying the extent of each association type, e.g., `authors-for-book(b1, $AUTHOR: author)` selects the authors of book `b1` (note the association role identifying the topic 'author'). tolog has only two "dynamic association predicates" similar to "dynamic occurrence predicates". The original version of tolog has only two "built-in predicates", `instance-of($INSTANCE, $CLASS)` and `direct-instance-of($INSTANCE, $CLASS)`, conveying the semantics of the subsumption hierarchy.

The current version of tolog [173,175] has further built-in predicates, e.g., `role-player` and `association-role`, for enumerating the associations, association roles, occurrences, and topics. These allow querying arbitrary topic maps without a-priori knowledge of the types used in the topic maps. Query 21 can only be implemented only using these predicates:

```
select $RELATED from
title($BOOK, "Bellum Civile"),
related($BOOK, $RELATED)?
related($X, $Y) :- {
role-player($R1, $X), association-role($A, $R1),
association-role($A, $R2), role-player($R2, $Y) |
related($X, $Z), related($Z, $Y)
}.
```

Conjunctions are expressed, as in Prolog, by commas. Disjunctions are in curly braces the disjuncts being separated by `|`.

The built-in predicates `instance-of` and `direct-instance-of` can indeed be implemented using tolog rules as follows [174]:

[23] However, the project page implemented in RDF Gateway is a show case.

[24] `http://www.ontopia.net/solutions/products.html`

[25] The prefix `i` serves to distinguish different identifiers.

```
direct-instance-of($INSTANCE, $CLASS) :-
i"http://psi.topicmaps.org/sam/1.0/#type−instance"(
$INSTANCE : i"http://psi.topicmaps.org/sam/1.0/#instance",
$CLASS : i"http://psi.topicmaps.org/sam/1.0/#class").
super-sub($SUB, $SUPER) :-
i"http://www.topicmaps.org/xtm/1.0/core.xtm#superclass−subclass"(
$SUB : i"http://www.topicmaps.org/xtm/1.0/core.xtm#subclass",
$SUPER : i"http://www.topicmaps.org/xtm/1.0/core.xtm#superclass").
descendant-of($DESC, $ANC) :- {
super-sub($DESC, $ANC) |
super-sub($DESC, $INT), descendant-of($INT, $ANC)
}.
instance-of($INSTANCE, $CLASS) :- {
direct-instance-of($INSTANCE, $CLASS) |
direct-instance-of($INSTANCE, $DCLASS), descendant-of($DCLASS, $CLASS)
}.
```

Negation is available, however its semantics in tolog is not yet specified [173]. tolog has constructs for aggregation and sorting (although deemed insufficient [173]), paged queries using `limit` and `offset` as in SQL, and a module concept. Thanks to tolog's (possibly recursive) rules, Queries 26 and 27 can be implemented in tolog.

Neither the formal semantics, nor the complexity of tolog have been investigated yet.

Project page:

`http://www.ontopia.net/omnigator/docs/query/tutorial.html`[26]

Implementations:

Ontopia Knowledge Suite: `http://www.ontopia.net/solutions/products.html`
Topic Maps toolkit TM4J: `http://tm4j.org/`

Online demonstrations:

Omnigator: `http://www.ontopia.net/`
`http://www.ontopia.net/omnigator/models/index.jsp`[27]

7.5.2 AsTMA?: Functional Style Querying of Topic Maps

AsTMa? [27, 25] is a functional query language in the style of XQuery [57]. AsTMa? offers several path languages for accessing data in topic maps. With AsTMa?, answers can be re-structured, yielding new XML documents.

Query 20 can be implemented as follows:

```
<books>
{ forall [$book (Writing)] in http://example.org/books
return
<book>
{$book,
forall $author in ($book →author / author-for-book) return
<author>
{$author}
<name>{$author/bn}</name>
</author>
</book> }
</books>
```

Query 20 can also be implemented as follows, using path expressions for accessing topics and associations:

[26] Tutorial.

[27] The demonstrator does not seem to support testing tolog queries.

```
<books>
{ forall [$book (Writing)] in http://example.org/books
return
<book>
{$book,
forall [ (author-for-book)
Writing : $book
author: $author ]
in http://example.org/books return
<author>
{$author}
<name>{$author/bn}</name>
</author>
</book> }
</books>
```

Project page:
`http://astma.it.bond.edu.au/querying.xsp`

Implementation):
As part of the Perl XTM module, available via CPAN

Online demonstration:
`http://astma.it.bond.edu.au/query/`

7.5.3 Toma: Querying Topic Maps inspired from SQL

Toma [237, 308] combines SQL syntax and path expressions for querying Topic Maps, i.e., the following query selects all books, specified as topics classified as `Writing`, with their authors:

```
select topic[book], topic[author]
from topic-type["Writing"].topic[book],
topic[book]..assoc[a]..topic[author],
assoc-type["author–for–book"].assoc[a]
```

Toma provides access to all Topic Maps concepts, including the subsumption hierarchy. Information about a topic such as topic identifier, basename, and subject identifier are accessed using the long name, or `.` notation, common in object-oriented languages, e.g., `$topic.bn = 'Julius Caesar'` compares the basename, short `bn`, of topics, short by `$topic`, with the string “Julius Caesar”. Associations can be traversed using `->`, predefined associations with special semantics, such as the instance-of and superclass-subclass associations, can be traversed transitively when traversing the subsumption hierarchy. `$start.super(1..∗)` selects all super-classes of the current class. Instead of `1..∗`, an interval, or a single number, can indicate how many superclass-subclass associations should be traversed. A similar notation is available for instance-of associations.

Query 20 can be expressed as follows:

```
select $book, $author, $author.bn
where $book.type(1..∗).id = 'Writing'
and author-for-book%a→Writing = $book
and author-for-book%a→author = $author
```

Query 22 can be expressed as follows:

```
select $topic
where $topic.type(1..∗).si.sir != 'http://example.org/books#Translator'
and not exists ($t.type(1) = $topic)
and not exists ($t.type(1..∗) = $x and $topic.super(1..∗) = $x)
```

This query selects all topics that are neither used as type of another topic, nor typed `Translator`. All topics are selected that neither (a) have the subject identifier `http://example.org/books#Translator`, nor (b) are the type of some topic, nor (c) are a sub-class of some topic that is some topic's type.

Project page:
`http://www.spaceapplications.com/toma/`

Implementation:
Not freely available

Online demonstration:
none

7.5.4 Path-based Access to Topic Maps: XTMPath and TMPath

Following the success of XPath, a number of path-based query languages have been proposed for Topic Maps, cf. [26] for an overview of a plea for the inclusion of path navigation in the upcoming ISO Topic Maps query language.

XTMPath [29] is a path-based query language relying on the XTM [306] serialisation of topic maps in XML. The following path selects all topics that are (directly) typed `Historical_Novel`:

```
topic[instanceOf/topicRef/@href = "\#Historical\_Novel"]}
```

This path expression reflects the XTM serialisation:

```
<topic id="b1">
<instanceOf> <topicRef xlink:href="#Historical_Novel"/> </instanceOf>
</topic>
```

Note that (1) Only a limited subset of the XPath constructs is supported by XTMPath, mostly the child and descendant axis and some simple predicates (in XPath's abbreviated syntax), and (2) XTMPath operates on data conforming to a single DTD[28], viz., the DTD of XTM DTD [29], leading to treating the axis "child" like the axis "descendant" with a few exceptions, e.g., `instanceOf`.

Project page:
`http://cpan.uwinnipeg.ca/htdocs/XTM/XTM/Path.html`

Implementation:
Available from CPAN as part of the XTM toolkit

Online demonstration:
none

7.6 Conclusion: Salient Aspects of the Query Languages Considered

This article is an attempt to give a survey of both query languages proposed for the "standard Web" (i.e., basically XML data), and query languages for the Semantic Web (i.e. mostly RDF and Topic Maps). Query

[28] Document Type Definition, cf. [71].

languages targeting OWL have not been considered in this survey, because as of writing (March 2005), they still are in their infancy and the few languages proposed so far can only query meta-data.

Inspite of the exclusions described in Section 7.1 (programming languages tools for XML, reactive languages for the Web, rules languages, and OWL query languages) a considerable number of languages have been considered in this article. Indeed, we are not aware of any other effort to survey Web and/or Semantic Web query languages at the same level of depth and breadth.

Even though the field is moving extremely fast and new proposals are always emerging, it is already possible and worthwhile to stress some of the salient aspects of Web and Semantic Web query languages:

Path vs. Logic or Navigational vs. Positional. Web and Semantic Web query languages express basic queries using one of two paradigms, paths à la XPath, or Logic, à la Logic Programming. These two paradigms can also be named "navigational" and "positional", respectively, stressing that (path-oriented) navigations inherently conflict with referential transparency. One might expect that both kinds of languages will continue to be investigated, yielding interesting opportunities for further comparison and research.

Logical Variables. When Web and Semantic Web query languages have variables, they almost always are logical variables, i.e., Logic Programming or Functional Programming variables, as opposed to variables in imperative programming languages that are amenable to explicit assignments.

Referential Transparency and (Weak or Strong) Answer-Closure. Referential Transparency (i.e., within the same scope, an expression always means the same), *the* trait of declarative languages, is, if not fully achieved, obviously striven for by both positional and logic, query languages, especially in Semantic Web query languages. Some query languages are "weakly answer-closed" or "answer-closed" in the sense of [107], i.e., they deliver answers in the formalism of the data queried. A few query languages are "strongly answer-closed", i.e., they make query programs possible that can further process data generated by these very programs. Arguably, strong answer-closure is important for structuring programs and sustaining the so-called "separation of concerns" in programming. One might expect that positional Web and Semantic Web query languages will mature into well-designed, referentially transparent and strongly answer-closed languages.

Backtracking-free Logic Programming or Set-Oriented Functional Query Evaluation. Positional, or logic query languages that offer construct similar to rules or views, are, with a few exceptions or unclear cases, backtracking-free. Equivalently, they can be called set-oriented functional. This convergence of two programming paradigms in Web query languages seems promising for further research.

Incomplete Queries and Answers. Many query languages offer means for incomplete specifications of queries, paying tribute to the "semi-structured" [5] nature of data on the Web, i.e., that data on the Web either has no schemas or does not fully respect its schema. Incomplete query specifications are extremely useful on the Semantic Web, too. In querying an RDF graph or topic maps, incomplete queries are very useful for easily accessing the neighbourhood of resources. Indeed such incomplete specifications considerably simplify and ease programming.

Versatile vs. Data Format Specific Query Languages. Most RDF query languages are RDF-specific, and even specifically designed for one serialisation. The authors are convinced that an evolution towards data format "versatile" languages that are capable of easily accommodating XML, RDF, Topic Maps, OWL, etc. without requiring "serialisation consciousness" from the programmer, should be striven for.

Reasoning Capabilities. Interestingly, but not surprisingly, not all XML query languages have views, rules, or similar concepts allowing the specification of other forms of reasoning. Surprisingly, the same holds true of RDF query languages. Many authors of RDF query languages see deduction and reasoning to be a feature of an underlying RDF store offering materialisation, i.e., completion of RDF data with derivable data prior to query evaluation. This is surprising, because one might expect many Semantic Web applications to access not only one RDF data store at one Web site, but instead many RDF data stores at different Web sites and to draw conclusions combining data from different stores. Such an RDF query scenario requires, on the decentralised and open Web, deduction at query time, i.e., when queries are evaluated.[29]

Language engineering. Language engineering issues, e.g., abstract data types and static type checking, modules, polymorphism, and abstract machines, have clearly not yet made their way in the Web query languages, as they did not in database query languages. This situation opens avenues for promising research of great practical, as well as theoretical relevance.

[29] Indeed, materialising conclusions from all possible combinations of Web sites is infeasible.

Chapter 8

Rich Clients need Rich Interfaces Query Languages for XML and RDF Access on the Web

A Tutorial at the German XML Tage 2006

To be held at the XML tage 2006 in Berlin, this tutorial is closely based on a revision of the material presented in this deliverable. It focuses on Web query languages for both server and client processing and covers both XML and RDF query languages.

Access to Web data has become an integral part of many applications and services. In the past, such data has usually been accessed through human-tailored HTML interfaces. Nowadays, rich client interfaces in desktop applications or, increasingly, in browser-based clients ease data access and allow more complex client processing based on XML or RDF data retrieved through Web service interfaces. Convenient specifications of the data processing on the client and flexible, expressive service interfaces for data access become essential in this context. Web query languages such as XQuery, XSLT, SPARQL, or Xcerpt have been tailored specifically for such a setting: declarative and efficient access and processing of Web data. This tutorial introduces, compares, and classifies the most relevant exemplars of Web query languages for XML, RDF, and/or TopicMaps data. Interesting features as well as differences in expressiveness and adequacy are digested along practical and concrete use cases. Emphasis is placed on recent W3C standardization activities, contrasted with alternative approaches from industry and academia.

8.1 Tutorial Details

Length of Tutorial:

3h (three hours). Giving the attendants an impression of current Web query languages and a set of guidelines for their use in developing and deploying Web applications requires a certain breadth for the tutorial. Despite a careful selection of highly relevant exemplars from the large number of languages considered in the underlying survey, a three hour time frame is necessary to cover the material in sufficient depth.

Intended Audience:

The topic is highly relevant for both practitioners and managers involved in the creation of modern Web applications that use Web service interfaces for data access. In this context, the tutorial serves as a decision help on how to provide flexible access to large data bases through query interfaces. In particular, the impact of query language concepts and features on the intended service are discussed. The tutorial is also relevant for experts in XML or Semantic Web technology as it provides a novel perspective over areas of research usually considered separate and introduces into many languages that are still largely unknown. However, though both the underlying survey and the tutorial itself provide ample pointers to more in-depth coverage of the languages considered, it is not intended as a comprehensive introduction into any of the languages discussed to the depth needed to be able to use them immediately.

Required Knowledge:

A basic understanding of Web technologies such as XML, HTTP, and Web Services is needed. Some knowledge of RDF and/or similar Semantic Web technologies is advantageous though not strictly necessary. Prior knowledge on query languages or Web query interfaces is *not* required though it certainly helps in some of the tutorial parts.

(Short) Tutorial Description:

Convenient specifications of Web data processing on the client and flexible, expressive service interfaces for data access on the server become essential in the context of Web services commoditizing structured information on the Web. Web query languages such as XQuery, XSLT, SPARQL, or Xcerpt have been tailored specifically for such a setting: declarative and efficient access and processing of Web data. This tutorial introduces, compares, and classifies the most relevant exemplars of Web query languages for XML, RDF, and/or TopicMaps data. The selection of these languages is based upon a recent survey of Web query languages [24] conducted by the authors. Interesting features as well as differences in expressiveness and adequacy are digested along practical and concrete use cases. Emphasis is placed on recent W3C standardization activities, contrasted with alternative approaches from industry and academia. The tutorial concludes with a discussion of advantages and challenges for deploying Web query languages today.

Overview/Outline of the Tutorial:

See following section.

Prior Presentations of this Tutorial:

An extended version of the tutorial (with intensive exercises) has been conducted (by James Bailey, François Bry, and Tim Furche) at the "Reasoning Web" 2005 Summer School, Mdina, Malta. A slightly shorter version has been presented (by Sebastian Schaffert) at the Franco-Mexican (Summer) School on Distributed Systems, Grenoble, France. The tutorial is based on an extensive survey of Web and Semantic Web query languages published as a chapter [24] in the Springer tutorial volume "Reasoning Web".

Technical Requirements:

The only technical requirements of the tutorial are common presentation equipment for the lecturers including an HTML browser for the slides. There are no technical requirements for the participants of the tutorial.

Note on Presenters:

This tutorial is based on work from all of the authors. At least two of the tutorial authors will present it, if accepted, at the "XML-Tage" 2006. The survey chapter [24] forms the basis of this tutorial but the material has been extended and updated continuously. This tutorial particularly considers also two recent developments: First, the ongoing activities on rule languages and rule interchange at the W3C and their relevance for Web query languages; second, the challenges for Web query languages arising from novel Web applications such as Google Maps, Flickr, or Zimbra, where the communication between Web browser and Web server moves more and more from "plain" HTML to (syntactically and semantically) "rich" XML and RDF data that can be processed by the client in more interesting ways than plain HTML.

Tutorial Language:

Based on the preferences of the conference organizers, the tutorial can be held either in English or in German language.

8.2 Outline of the Tutorial

In this tutorial, a large number of Semantic Web query languages are considered with focus on those (a) already seeing wide-spread adoption, (b) under consideration for standardization, or (c) providing novel and influential perspectives on Semantic Web querying.

The languages considered can roughly by divided in three groups based on the format of the data queried (XML, RDF, or TopicMaps). Additionally, the tutorial also gives a brief outlook into two very recent research directions: rules and rule interchange in Web query languages and versatile Semantic Web query languages that allow intertwined access to data in different representation formalisms, e.g., to RDF and Topic Maps data. The actual discussion of the query languages will therefore be oriented on the following structure:

1. **Introduction, Sample Data, and Query Scenario** (15 minutes)

2. **XML Query Languages** (60 minutes)

   (a) "*W3C's Query Languages: The Navigational Approach*" introduces the mainstream XML query languages XPath, XQuery, XSLT. Emphasis is placed on foundational principles and recent developments.

   (b) "*The Positional Approach to XML Querying: An Better Way?*" surveys a group of alternative query languages from academia that suggest a different approach for XML querying similar to the "Query-by-Example" [379] paradigm for relational data.

3. **RDF Query Languages** (60 minutes):

   (a) "*The SPARQL family*" presents the family of languages originating from the proposal of SquishQL [282]. The focus lies on RDQL [332, 332], a widely adopted RDF query language implemented, e.g., in HP's Jena Toolkit [190], and on SPARQL [317], the RDF query language currently under standardization at the W3C.

   (b) "*The RQL family*" discusses the RQL [218] language, a very different perspective on RDF querying than SPARQL focused on a strong type system and rich, but complex language constructs. Several proposals for simplifying RQL, e.g., SeRQL and eRQL, highlight strength and weaknesses of this approach.

   (c) "*Navigational Access to RDF: Versa*" presents the only exemplar of an RDF query language that uses the navigational access dominating the mainstream XML query languages.

(d) "*Reactive Rules in RDF Query Languages*" discusses reactive rules and their use in some prototypical RDF query languages, e.g., Algae [312], the query language of the W3C Annotea project, iTQL [1] and Nokia's WQL [244], two industry proposals for RDF query languages.

(f) "*Deductive Query Languages.*" A final pitch on RDF query languages is provided by query languages in logic-programming style. TRIPLE [342] is an early proposal based on frame logic. Xcerpt [329, 328] is a query language developed by some of the authors of this tutorial that integrates RDF querying with access to other data formats, especially XML, and provides Prolog-style reasoning.

4. **Topic Maps Query Languages** (15 minutes):

   In contrast to RDF, the number of query languages for Topic Maps is still rather small. Therefore this part of the tutorial only briefly skims over the most important developments including the current standardization activities at ISO that are expected to result in a standard Topic Maps Query Language.

5. Outlook on **Rules and Web Querying** (10 minutes):

   Many of the above mentioned languages have rules or similar mechanisms, e.g., for integrating Web data, basic reasoning, or reactivity. The W3C is currently starting an activity to standardize rule languages and rule interchange on the Web. Goals and charter of this activity, as well as its relevance for the larger topic of this tutorial, are briefly summarized.

6. Outlook on **Versatile Web Query Languages** (10 minutes):

   The vision of versatile Web query languages will be introduced and exemplified using Xcerpt [329, 328].

7. **Summary and Conclusion** (10 minutes):

   The discussion of the query languages closes with a summary and comparison of central language features.

Chapter 9

RDF Querying: Language Constructs and Evaluation Methods Compared

A Survey for the REWERSE Reasoning Web 2006 Summer School

For the 2006 REWERSE Reasoning Web 2006 summer school, a course on RDF query languages, illuminating language constracts, and evaluation methods is being prepared at the moment. The course material is based on a thorough revision and extension of the RDF part of I4-D1 that (1) updates the material where necessary, (2) gives a more in-depth consideration to SPARQL, the soon-to-be-finished W3C recommendation for RDF querying, (3) discusses a list of language constructs that are desirable or challenging in RDF query languages, and (4) introduces into the current state of the art in evaluation and optimization techniques for these languages.

During the last two years, a plethora of query languages for RDF have been proposed. These languages can be grouped into the following "families": The "SPARQL family", including SquishQL, RDQL, RDFQL, SPARQL, and TriQL; the "RQL family" including RQL, SeRQL, and eRQL; the "XQuery inspired family" including the so-called "syntactic Web approach" to RDF querying and XsRQL; the "XSLT inspired family" including TreeHugger, RDF Twig, RDFT, and the query language of Nexus; the "XPath inspired family" including Versa, RDF Path, RPath, RxPath, and RxSLT; the "Controlled English family" currently with only Metalog; the "reactive rule family" including Algae, iTQL, WQL; the "deductive rule family" including N3QL, R-DEVICE, TRIPLE, and Xcerpt; the "QBE inspired family" including RDF-QBE, RDFQL, and visXcerpt. The lecture introduces into these families and their languages. Then, compares these families and languages considering first the constructs, second the evaluation methods, and third the reasoning capabilities of the languages. Concerning the language constructs, the capability to express grouped selection of RDF data, optional selections, triple-based vs. path-based data selection are considered. Concerning query evaluation, the capability to access RDF data at sites retrieved from partial answers, to cope with non-trivial cases, including cyclic dependencies and/or data, and to traverse (arbitrary length) paths in the RDF graph efficiently are considered. Concerning reasoning, the capability to derive data implied by the RDF and RDFS semantics and by user defined rules are considered.

Bibliography

[1] iTQL Commands. Online only, 2004.

[2] RDFQL Database Command Reference. Online only, 2004.

[3] Langdale Consultants . Nexus Query Language. Online only, 2000.

[4] S. Abiteboul. Querying Semi-Structured Data. In *Proc. International Conference on Database Theory*, 1997.

[5] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries1(1):68-88, April 1997.*, 1(1):68–88, 1997.

[7] S. Abiteboul and V. Vianu. Queries and Computation on the Web. In *Proc. International Conference on Database Theory*, pages 262–275, 1997.

[8] *Adobe Extensible Metadata Platform (XMP)*, 2001, Adobe Systems Inc.

[9] S. Alexaki, N. Athanasis, V. Christophides, G. Karvounarakis, A. Maganaraki, D. Plexousakis, and K. Toll. The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web. *ERCIM News*, (51 (Special Issue on the Semantic Web)), 2002.

[10] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proc. International Workshop on the Semantic Web*, 2001.

[11] J. Alferes, W. May, and P. Patranjan. *State of the Art on Evolution and Reactivity*, 2004, Deliverable.

[12] G. Alliance. RDF::Core::Query—Implementation of Query Language. Online only, 2004.

[13] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, D. McBeath, M. Rys, and J. Shanmugasundaram. *XQuery and XPath Full-Text*, 2004, W3C, Working Draft. Available from: `http://www.w3.org/TR/xquery-full-text-requirements/`.

[14] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In *Proc. Int. World Wide Web Conf.*, 2004.

[15] S. Amer-Yahia, M. F. Fernandez, D. Srivastava, and Y. Xu. PIX: Exact and Approximate Phrase Matching in XML. In *Proc. ACM SIGMOD Conf.*, 2003.

[16] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *Proc. ACM SIGMOD Conf.*, 2004.

[17] C. Anutariya, V. Wuwongse, and V. Wattanapailin. An Equivalent-Transformation-Based XML Rule Language. In *Proc. International Workshop on Rule Markup Languages for Business Rules in the Semantic Web*, 2002.

[18] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification. Recommendation, W3C, 10 1998.

[19] E. Augurusa, D. Braga, A. Campi, and S. Ceri. Design and Implementation of a Graphical Interface to XQuery. In *Proc. Symposium of Applied Computing*, pages 1163–1167. ACM Press, 2003.

[20] D. Backett. New Syntaxes for RDF. Online only, November 2003.

[21] D. Backett. Modernising Semantic Web Markup. In *Proc. XML Europe*, April 2004.

[22] E. Bae and J. Bailey. CodeX: an approach for debugging XSLT transformations. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, 2003.

[23] J. Bailey. Transformation and Reaction Rules for Data on the Web. In *Proc. Australasian Database Conference*, 2005.

[24] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In J. Maluszinsky and N. Eisinger, editors, *Reasoning Web Summer School 2005*, pages 35–133. Springer-Verlag, LNCS 3564, 2005.

[25] R. Barta. AsTMa? Tutorial. Technical report, Bond University, 2003.

[26] R. Barta. Path Language for Topic Maps: Full speed ahead? Online only, 2004.

[27] R. Barta. AsTMa= Language Definition. Online only, 2007.

[28] R. Barta and L. Garshol. *Topic Map Query Language, Use Cases*, December 2003, ISO/IEC, Technical document. Available from: `http://www.y12.doe.gov/sgml/sc34/document/0449.htm`.

[29] R. Barta and J. Gylta. *XTM::Path*, 2002.

[30] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, and P. Velikhov. XML-Based Information Mediation with MIX. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1999.

[31] N. Bassiliades and I. Vlahavas. Intelligent Querying of Web Documents Using a Deductive XML Repository. In *Proc. Hellenic Conference on Artificial Intelligence*, April 2002.

[32] N. Bassiliades and I. Vlahavas. Capturing RDF Descriptive Semantics in an Object Oriented Knowledge Base System. In *Proc. International Word Wide Web Conference*, May 2003.

[33] R. Baumgartner, S. Flesca, and G. Gottlob. The Elog Web Extraction Language. In *Proc. International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, December 2001.

[34] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki. An Evaluation of Binary XML Encoding Optimizations for fast Stream based XML Processing. In *Proc. Int. World Wide Web Conf.*, pages 345–354. ACM Press, 2004.

[35] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. *OWL Web Ontology Language—Reference*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/owl-ref/`.

[36] D. Beckett. A retrospective on the development of the RDF/XML Revised Syntax. Online only, June 2003.

[37] D. Beckett. *Turtle - Terse RDF Triple Language*, February 2004.

[38] D. Beckett and B. McBride. *RDF/XML Syntax Specification (Revised)*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/rdf-syntax-grammar/`.

[39] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *Proc. International Conference on Database Theory*, 2003.

[40] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proc. International Conference on Functional Programming*, 2003.

[41] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web. In *Proc. Int. Semantic Web Conf.*, 11 2004. I4 I3.

[42] S. Berger, F. Bry, and S. Schaffert. A Visual Language for Web Querying and Reasoning. In *Proc. Workshop on Principles and Practice of Semantic Web Reasoning*, LNCS 2901. Springer-Verlag, December 2003.

[43] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Proc. Int. Conf. on Very Large Databases*, 2003.

[44] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. *XML Path Language (XPath) 2.0*, 2005, W3C, Working Draft.

[45] A. Berlea and H. Seidl. fxt—A Transformation Language for XML Documents. *Journal of Computing and Information Technology, Special Issue on Domain-Specific Languages*, 2001.

[46] A. Berlea and H. Seidl. Binary Queries for Document Trees. *Nordic Journal of Computing*, 11(1):41–71, 2004.

[47] T. Berners-Lee. A strawman Unstriped syntax for RDF in XML. Online only, May 1999.

[48] T. Berners-Lee. N3QL—RDF Data Query Language. Online only, 2004.

[49] T. Berners-Lee. Notation 3, an RDF language for the Semantic Web. Online only, 2004.

[50] T. Berners-Lee. Primer: Getting into RDF and Semantic Web using N3. Online only, 2004.

[51] T. Berners-Lee. Semantic Web Road Map. Online only, 2004.

[52] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web—A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 2001.

[53] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[54] P. Biron and A. Malhotra. *XML Schema Part 2: Datatypes*, 2001, W3C, Recommendation. Available from: `http://www.w3.org/TR/xmlschema-2/`.

[55] C. Bizer. The TriG Syntax. Online only, April 2004.

[56] C. Bizer. TriQL—A Query Language for Named Graphs. Online only, 2004.

[57] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. *XQuery 1.0: An XML Query Language*, 2005, W3C, Working Draft.

[58] S. Boag, D. Chamberlin, M. F. FernÃąndez, D. Florescu, J. Robie, and J. SimÃľon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2 2005.

[59] D. Bogachev. TMPath – Introduction. Online only, 2003.

[60] D. Bogachev. TMPath – Revisited. Online only, 2004.

[61] H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. RuleML Design. Online only, 2002.

[62] O. Bolzer. Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/Master thesis, University of Munich, 2 2005.

[63] O. Bolzer, F. Bry, T. Furche, S. Kraus, and S. Schaffert. Development of Use Cases, Part I: Illustrating the Functionality of a Versatile Web Query Language. Deliverable I4-D3, REWERSE, 3 2005. I4.

[64] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. Int. Conf. on Data Engineering*, page 403. IEEE Computer Society, 2002.

[65] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *ACM SIGMOD Record*, 2000.

[66] A. Bonifati and D. Lee. Technical Survey of XML Schema and Query Languages. Technical report, University of California, Los Angeles, Computer Science Dept., 2001.

[67] D. Braga, A. Campi, S. Ceri, and E. Augurusa. XQuery by Example. In *Proc. Int. World Wide Web Conf.*, 2003.

[68] T. Bray. RPV: Triples Made Plain. Online only, 2002.

[69] T. Bray, D. Hollander, and A. Layman. *Namespaces in XML*, 1999, W3C, Recommendation. Available from: `http://www.w3.org/TR/REC-xml-names/`.

[70] T. Bray, D. Hollander, A. Layman, and R. Tobin. *Namespaces in XML 1.1*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/xml-names11/`.

[71] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition)*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/REC-xml/`.

[72] J.-M. Bremer and M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In *Int. Workshop on the Web and Databases*, 2002.

[73] D. Brickley. RDF: Understanding the Striped RDF/XML Syntax. Online only, October 2001.

[74] D. Brickley, R. Guha, and B. McBride. *RDF Vocabulary Description Language 1.0: RDF Schema*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/rdf-schema/`.

[75] D. Brickley and L. Miller. *FOAF Vocabulary Specification*, May 2004.

[76] J. Broekstra, C. Fluit, and F. van Harmelen. The State of the Art on Represenation and Query Languages for Semistructured Data. On-To-Knowledge EU-IST-1999-10132 Deliverable 8, Aidministrator Nederland b.v., 2000.

[77] J. Broekstra and A. Kampman. Query Language Definition. On-To-Knowledge EU-IST-1999-10132 Deliverable 9, Aidministrator Nederland b.v., 2001.

[78] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003.

[79] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. International Semantic Web Conference*, 2002.

[80] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[81] E. Bruno, J. L. Maitre, and E. Murisasco. Extending XQuery with Transformation Operators. In *Proc. ACM symposium on Document Engineering*, pages 1–8. ACM Press, 2003.

[82] F. Bry, W. Drabent, and J. Maluszynski. On Subtyping of Tree-structured Data A Polynomial Approach. In *Proc. Workshop on Principles and Practice of Semantic Web Reasoning, St. Malo, France*, volume 3208 of *LNCS*. REWERSE, Springer-Verlag, 9 2004. I4 I3.

[83] F. Bry, T. Furche, L. Badea, C. Koch, S. Schaffert, and S. Berger. Identification of Design Principles for a (Semantic) Web Query Language. Deliverable I4-D1, REWERSE, 2004.

[84] F. Bry, T. Furche, L. Badea, C. Koch, S. Schaffert, and S. Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems*, 1(2), 2005. I4.

[85] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. Symposium of Applied Computing*. ACM, 3 2005. I4 I5.

[86] F. Bry, P.-L. Pătrânjan, and S. Schaffert. Xcerpt and XChange: Logic Programming Languages for Querying and Evolution on the Web. In *Proc. Int. Conf. on Logic Programming*, LNCS, 2004.

[87] F. Bry and S. Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002.

[88] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Proc. Int. Workshop on Web and Databases*, volume 2593 of *LNCS*. Springer-Verlag, 2002.

[89] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. Int. Conf. on Logic Programming*, volume 2401 of *LNCS*. Springer-Verlag, 7 2002.

[90] F. Bry, S. Schaffert, and A. Schröder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Proc. Workshop Logische Programmierung*, March 2004.

[91] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. ACM SIGMOD Conf.*, pages 505–516. ACM Press, 1996.

[92] P. Buneman, S. B. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proc. Int. Workshop on Database Programming Languages*, page 12. Springer-Verlag, 1996.

[93] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[94] S. Buxton and M. Rys. *XQuery and XPath Full-Text—Requirements*, 2004, W3C, Working Draft. Available from: `http://www.w3.org/TR/xquery-full-text-requirements/`.

[95] A. b.v. and S. A. Ltd. *The SeRQL query language*, chapter 5. Aduna b.v., Sirma AI Ltd., 2002.

[96] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of Conjunctive Regular Path Queries with Inverse. In *Proc. Int. Conf. on the Principles of Knowledge Representation and Reasoning*, pages 176–185, 2000.

[97] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Query Processing using Views for Regular Path Queries with Inverse. In *Proc. ACM Symposium on Principles of Database Systems*, pages 58–66, 2000.

[98] L. Cardelli. *Type systems*, chapter 103, pages 2208–2236. ACM Press, 1996.

[99] L. Cardelli and G. Ghelli. TQL: a Query Language for Semistructured Data based on the Ambient Logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.

[100] L. Cardelli and A. D. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *Proc. Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, 2000.

[101] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named Graphs, Provenance and Trust. Technical Report HPL-2004-57, HP Labs, 2004.

[102] J. Carroll and P. Stickler. TriX: RDF Triples in XML. Online only, May 2004.

[103] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[104] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca:. XML-GL: A Graphical Language for Querying and Reshaping XML Documents. In *Proc. W3C QL'98 – Query Languages*, 1998.

[105] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. TancaÂă. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. In *Proc. Int. World Wide Web Conf.*, 1999.

[106] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer Verlag, 1990.

[107] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Cases*, 2005, W3C, Working Draft.

[108] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. *XML Query (XQuery) Requirements*, 2003, W3C, Working Draft.

[109] D. Chamberlin and J. Robie. XQuery Update Facility Requirements. Working draft, W3C, 2005.

[110] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. Workshop on Web and Databases*, 2000.

[111] A. Chandra and D. Harel. Computable queries for relational data bases (Preliminary Report). *Journal of Computer Systems and Sciences*, 21:156–178, 1980.

[112] A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer Systems and Sciences*, 25(1):99–128, 1982.

[113] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *Proc. Workshop on the Web and Databases*, 2002.

[114] Y. Chen and P. Revesz. CXQuery: A Novel XML Query Language. In *Proc. International Conference on Advances in Infrastructure for Electronic Business, Science, and Medicine on the Internet*, 2002.

[115] Z. Chen, H. V. Jagadish, L. V. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proc. Int. Conf. on Very Large Databases*, 2003.

[116] T. T. Chinenyanga and N. Kushmerick. An Expressive and Efficient Language for XML Information Retrieval. *Journal of the American Society for Information Science and Technology*, 53(6):438–453, 2002.

[117] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 413–422, 1996.

[118] V. Christophides, G. Karvounarakis, I. Koffina, G. Kokkinidis, A. Magkanaraki, D. Plexousakis, G. Serfiotis, and V. Tannen. The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware. In *Proc. International Workshop on Semantic Web and Databases*, 2003.

[119] V. Christophides, D. Plexousakis, G. Karvounarakis, and S. Alexaki. Declarative Languages for Querying Portal Catalogs. In *Proc. DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[120] J. Clark. *XSL Transformations (XSLT) Version 1.0*, 1999, W3C, Recommendation.

[121] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*, 1999, W3C, Recommendation.

[122] J. Clark and M. Makoto. *RELAX NG Specification*, 2001, OASIS, Committee Specification.

[123] J. Clark and M. Makoto. *Regular-grammar-based validation—RELAX NG*, 2002, International Organisation for Standardization, Draft International Standard.

[124] K. Clark. *RDF Data Access Use Cases and Requirements*, 2004, W3C, Working Draft.

[125] K. Clark and D. Connolly. RDF Data Access Design Evaluations. Online only, 2004.

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

[127] E. Codd. Relational Completeness of Data Base Sublanguages. In R. Rustin, editor, *Database Systems: Courant Computer Science Symposia 6*. Prentice-Hall, 1972.

[128] J. Coelho and M. Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Proc. Int. Conf. on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems*, volume 3291 of *LNCS*. Springer-Verlag, 2004.

[129] S. Cohen, Y. Kanza, Y. Kogan, Y. Sagiv, W. Nutt, and A. Serebrenik. EquiX—a search and query language for XML. *Journal of the American Society for Information Science and Technology*, 53(6):454–466, 2002.

[130] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *Proc. Int. Conf. on Very Large Databases*, 2003.

[131] S. Comai, E. Damiani, and P. Fraternali. Computing Graphical Queries over XML Data. *ACM Transactions on Information Systems*, 19(4):371–430, 2001.

[132] S. Comai, S. Marrara, and L. TancaÂă. XML Document Summarization: Using XQuery for Synopsis Creation. In *Proc. Int. Workshop on Database and Expert Systems Applications*, 2004.

[133] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. The Query Language TQL. In *Proc. Int. Workshop on the Web and Databases*, 2002.

[134] J. Cowan and R. Tobin. *XML Information Set (Second Edition)*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/2004/REC-xml-infoset-20040204/`.

[135] R. Daniel. *Harvesting RDF Statements from XLinks*, 2000, W3C, Note.

[136] I. Davis. RDF Template Language 1.0. Online only, September 2003.

[137] N. Deakin. ReoPath. Online only, 2003.

[138] S. Decker, D. Brickley, J. Saarela, and J. Angele. A Query and Inference Service for RDF. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[139] D. DeHaan, D. Toman, M. P. Consens, and M. T. ÃŰzsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. ACM SIGMOD Conf.*, pages 623–634. ACM Press, 2003.

[140] S. DeRose, R. Daniel, E. Maier, and J. Marsh. *XPointer xmlns() Scheme*, 2003, W3C, Recommendation. Available from: `http://www.w3.org/TR/xptr-xmlns/`.

[141] S. DeRose, E. Maier, and R. Daniel. *XPointer xpointer() Scheme*, 2002, W3C, Working Draft. Available from: `http://www.w3.org/TR/xptr-xpointer/`.

[142] S. DeRose, E. Maier, and D. Orchard. *XML Linking Language (XLink) Version 1.0*, 2001, W3C, Recommendation. Available from: `http://www.w3.org/TR/xlink/`.

[143] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *XML-QL: A Query Language for XML*, 1998, W3C, Note. Available from: `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[144] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *Proc. W3C QL'98 – Query Languages 1998*. W3C, 1998.

[145] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proc. Int. World Wide Web Conf.*, 1999.

[146] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[147] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proc. Int. Workshop on Knowledge Representation meets Databases*, 2001.

[148] C. Dollin. Jena Toolkit—Reification Howto. Online only, 2003.

[149] C. Dong and J. Bailey. Optimization of XML Transformations Using Template Specialization. In *Proc. Int. Conf. on Web Information Systems Engineering*, 2004.

[150] C. Dong and J. Bailey. Static Analysis of XSLT Programs. In *Proc. Australasian Database Conf.*, pages 151–160. Australian Computer Society, Inc., 2004.

[151] D. Draper, P. Frankhauser, M. FernÃąndez, A. Malhotra, K. Rose, M. Rys, J. SimÃľon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Working draft, W3C, 2 2005.

[152] E. Dumbill. Putting RDF to Work. Online only, 2000.

[153] D. Eastlake and A. Panitz. Reserved Top Level DNS Names. RFC 2606, IETF, 1999.

[154] A. Eisenberg and J. Melton. An early Look at XQuery. *SIGMOD Record*, 31(4):113–120, 2002.

[155] A. Eisenberg and J. Melton. An early Look at XQuery API for Java™(XQJ). *SIGMOD Record*, 33(2):105–111, 2004.

[156] D. Fallside. *XML Schema Part 0: Primer*, 2001, W3C, Recommendation. Available from: `http://www.w3.org/TR/xmlschema-0/`.

[157] P. Fankhauser. XQuery Formal Semantics: State and Challenges. *SIGMOD Record*, 30(3):14–19, 2001.

[158] P. Fankhauser and P. Lehti. XQuery by the book: The IPSI XQuery Demonstrator. In *XML Conference & Exhibition*, 2002.

[159] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model*, 2004, W3C, Working Draft.

[160] M. Fernandez, J. Simeon, and P. Wadler. *XML Query Languages: Experiences and Exemplars*, 1999, Draft. Available from: `http://www.w3.org/1999/09/ql/docs/xquery.html`.

[161] M. FernÃąndez, J. SimÃľon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0 : The Galax Experience. In *Proc. Int. Conf. on Very Large Databases*, 2003.

[162] R. Fikes, P. Hayes, and I. Horrocks. *DAML Query Language (DQL): Abstract Specification*, 2002, DAML Joint Committee.

[163] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL – A Language for Deductive Query Answering on the Semantic Web. *Journal of Web Semantics*, To appear.

[164] D. Florescu, M. Fernandez, A. Levy, and D. Suciu. A Query Language and Processor for a Web-site Management System. In *Proc. Workshop on Management of Semi-structured Data*, 1997.

[165] D. Florescu, A. GrÂ§nhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proc. International World Wide Web Conference*, May 2002.

[166] D. Florescu, A. GrÂ§nhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. *Computer Networks*, 42(5), 2003.

[167] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[168] D. Florescu, A. Levy, M. Fernandez, and D. Suciu. A Query Language for a Web-site Management System. *SIGMOD Record*, 26(3):4–11, 1997.

[169] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In *Proc. International Conference on Very Large Databases*, 1994.

[170] N. Fuchs, U. Schwertel, and R. Schwitter. Attempto Controlled English (ACE) Language Manual, Version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich, 1999.

[171] N. Fuhr and K. Gross. XIRQL: a Query Language for Information Retrieval in XML Documents. In *Proc. ACM Conference on Research and Development in Information Retrieval*, 2001.

[172] L. Garshol. tolog—A topic map query language. In *Proc. XML Europe*, 2001.

[173] L. Garshol. Extending tolog—Proposal for tolog 1.0. In *Proc. Extreme Markup Languages*, 2003.

[174] L. Garshol. tolog 0.1. Technical report, Ontopia, 2003.

[175] L. Garshol. tolog–Language tutorial. Online only, 2004.

[176] L. Garshol. The Linear Topic Map Notation. Online only, 2007.

[177] L. Garshol and R. Barta. *Topic Map Query Language, Requirements*, November 2003, ISO/IEC, Draft. Available from: `http://www.y12.doe.gov/sgml/sc34/document/0448.htm`.

[178] L. M. Garshol. Living with Topic Maps and RDF. Online only, 2003.

[179] B. Glimm. A Query Language for Web Ontologies. Bachelor thesis, Hamburg University of Applied Sciences, University of Manchester, 2004.

[180] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A Standard Textual Interchange Format for the Object Exchange Model (OEM). Technical report, Database Group, Stanford University, 1996.

[181] G. Gottlob and C. Koch. Monadic Queries over Tree-Structured Data. In *Proc. Annual IEEE Symposium on Logic in Computer Science*, pages 189–202. IEEE Computer Society, 2002.

[182] G. Gottlob and C. Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In 51, editor, *Journal of the ACM*, volume 1, pages 74–113, 2004.

[183] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. International Conference on Very Large Databases*, 2002.

[184] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proc. ACM Symposium on Principles of Database Systems*, 2003.

[185] G. Gottlob, C. Koch, and R. Pichler. XPath Query Evaluation: Improving Time and Space Efficiency. In *Proc. International Conference on Data Engineering*, 2003.

[186] J. Grant and D. Backett. *RDF Test Cases*, February 2004, W3C.

[187] S. Groppe and S. BÃűttcher. XPath Query Transformation based on XSLT Stylesheets. In *Proc. Int. Workshop on Web Information and Data Management*, pages 106–110. ACM Press, 2003.

[188] P. Grosso, E. Maier, J. Marsh, and N. Walsh. *XPointer element() Scheme*, 2003, W3C, Recommendation. Available from: `http://www.w3.org/TR/xptr-element/`.

[189] P. Grosso, E. Maier, J. Marsh, and N. Walsh. *XPointer Framework*, 2003, W3C, Recommendation. Available from: `http://www.w3.org/TR/xptr-framework/`.

[190] H. L. S. W. R. Group. Jena – A Semantic Web Framework for Java. Online only, 2004.

[191] T. Grust. Accelerating XPath Location Steps. In *Proc. ACM SIGMOD Conf.*, 2002.

[192] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004.

[193] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[194] R. Guha. *Contexts: A Formalization and Some Applications*. PhD thesis, Stanford University, 1995.

[195] R. Guha. rdfDB Query Language. Online only, 2000.

[196] R. Guha, O. Lassila, E. Miller, and D. Brickley. Enabling Inferencing. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[197] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proc. ACM SIGMOD Conf.*, 2003.

[198] Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evaluation. In *Proc. Asia Pacific Web Conference*, 2004.

[199] J. Gylta. XTMPath, Manipulating Topic Map Data Structures. Online only, 2002.

[200] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *Proc. International Semantic Web Conference*, 2004.

[201] M. Harren, M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML Processing into Java. In *Proc. International World Wide Web Conference*, 2004.

[202] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proc. International Workshop on Practical and Scalable Semantic Systems*, 2003.

[203] A. Harth. Triple Tutorial. Online only, 2004.

[204] P. Hayes and B. McBride. *RDF Semantics*, 2004, W3C, Recommendation. Available from: http://www.w3.org/TR/rdf-mt/.

[205] J. Hidders. Satisfiability of XPath Expressions. In *Int. Workshop on Databse Programming Languages*, 2003.

[206] I. Horrocks and P. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. In *Proc. International Semantic Web Conference*, pages 17–29. Springer-Verlag, 2003.

[207] I. Horrocks and P. Patel-Schneider. A Proposal for an OWL Rules Language. In *Proc. International World Wide Web Conference*, pages 723–731. ACM, 2004.

[208] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. *SWRL: A Semantic Web Rule Language—Combining OWL and RuleML*, 2004, W3C, Member submission. Available from: http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

[209] I. Horrocks and S. Tessaris. A Conjunctive Query Language for Description Logic ABoxes. In *Proc. National Conference on Artificial Intelligence*, 2000.

[210] I. Horrocks, F. van Harmelen, and P. Patel-Schneider. *DAML+OIL*, 2001, Joint US/EU ad hoc Agent Markup Language Committee, Revised Language Specification. Available from: http://www.daml.org/2001/03/daml+oil-index.html.

[211] H. Hosoya and B. Pierce. XDuce: A Typed XML Processing Language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[212] J. Hynynen and O. Lassila. On the Use of Object-Oriented Paradigm in a Distributed Problem Solver. *AI Communications*, 2(3):142–151, 1989.

[213] T. D. C. M. Initiative. DCMI term declarations represented in RDF schema language. Online only, March 2003.

[214] Intellidimension. RDF Gateway. Online only, 2004.

[215] *ISO/IEC 13250 Topic Maps*, 1999, International Organization for Standardization, International Standard. Available from: http://www.y12.doe.gov/sgml/sc34/document/0322_files/iso13250-2nd-ed-v2.pdf.

[216] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT Programs to Efficient SQL Queries. In *Proc. Int. World Wide Web Conf.*, pages 616–626. ACM Press, 2002.

[217] B. Johnson and B. Shneiderman. Tree-maps: a Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proc. Int. Conf.on Visualization*, pages 284–291, 1991.

[218] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proc. International World Wide Web Conference*, May 2002.

[219] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF Descriptions for Community Web Portals. In *Proc. Journees Bases de Donnees Avancees*, 2001.

[220] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. Querying the Semantic Web with RQL. *Computer Networks and ISDN Systems Journal*, 42(5):617–640, August 2003.

[221] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. RQL: A Functional Query Language for RDF. In P. Gray, P. King, and A. Poulovassilis, editors, *The Functional Approach to Data Management*, chapter 18, pages 435–465. Springer-Verlag, 2004.

[222] H. Katz. XsRQL: an XQuery-style Query Language for RDF. Online only, 2004.

[223] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 1st edition, 8 2003.

[224] M. Kay. *XPathÂă2.0 Programmer's Reference*. John Wiley, 8 2004.

[225] M. Kay. *XSLT 2.0 Programmer's Reference*. John Wiley, 3rd edition, 8 2004.

[226] M. Kay. XSLT and XPath Optimization. In *XML Europe*, 2004.

[227] M. Kay. *XSL Transformations (XSLT) Version 2.0*, 2005, W3C, Working Draft.

[228] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 Serialization. Working draft, W3C, 2 2005.

[229] S. Kepser. A Simple Proof of the Turing-Completeness of XSLT and XQuery. In *Proc. Extreme Markup Languages*, 2004.

[230] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object Oriented and Frame Based Languages. *Journal of ACM*, 42:741–843, 1995.

[231] C. Kirchner, Z. Oian, P. Singh, and J. Stuber. Xemantics: a Rewriting Calculus-Based Semantics of XSLT. Technical Report A01-R-386, LORIA, 2002.

[232] G. Klyne. Contexts for RDF Information Modelling. Online only, 2000.

[233] G. Klyne and J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*, February 2004, W3C.

[234] G. Klyne, J. Carroll, and B. McBride. *Resource Description Framework (RDF): Concepts and Abstract Syntax*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/rdf-concepts/`.

[235] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[236] S. Kraus. Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web. Diplomarbeit/Master thesis, University of Munich, 2004.

[237] R. Ksiezyk. Answer is just a question [of matching Topic Maps]. In *Proc. XML Europe*, 2000.

[238] M. Lacher and S. Decker. On the Integration of Topic Maps and RDF Data. In *Proc. Extreme Markup Languages*, 2001.

[239] M. Lacher and S. Decker. RDF, Topic Maps, and the Semantic Web. *Markup Languages: Theory and Practice*, 3(3):313–331, December 2001.

[240] O. Lassila. BEEF Reference Manual—A Programmer's Guide to the BEEF Frame System, Second Version. Technical Report HTKK-TKO-C46, Department of Computer Science, Helsinki University of Technology, 1991.

[241] O. Lassila. Enabling Semantic Web Programming by Integrating RDF and Common Lisp. In *Proc. Semantic Web Working Symposium*, july 2001.

[242] O. Lassila. Taking the RDF Model Theory Out for a Spin. In *Proc. Semantic Web Working Symposium*, June 2002.

[243] O. Lassila. Ivanhoe: an RDF-Based Frame System. Online only, 2004.

[244] O. Lassila. Wilbur Query Language Comparison. Online only, 2004.

[245] O. Lassila. Wilbur Semantic Web Toolkit. Online only, 2004.

[246] O. Lassila and R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*, 1999, W3C, Recommendation. Available from: `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[247] A. Laux and L. Martin. *XUpdate—XML Update Language*, 2000, XML:DB Initiative, Working Draft. Available from: `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`.

[248] J. Liu and M. Vincent. Query translation from XSLT to SQL. In *Proc. Int. Database Engineering and Applications Symposium*, 2003.

[249] M. Liu. A Logical Foundation for XML. In *Proc. International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 2002.

[250] B. Ludäscher, R. Himmeroeder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-oriented Perspective. *Information Systems*, 23(8):1–25, 1998.

[251] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. *A Brief Introduction to XMAs*, 1999, Database Group, University of California, San Diego.

[252] R. Luk, H. Leong, T. Dillon, A. Chan, W. B. Croft, and J. Allan. A Survey in Indexing and Searching XML Documents. *Journal of the American Society for Information Science and Technology*, 53(6):415–437, 2002.

[253] A. Magkanaraki, G. Karvounarakis, V. Christophides, D. Plexousakis, and T. Anh. Ontology Storage and Querying. Technical Report 308, Foundation for Research and Technology Hellas, April 2002.

[254] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proc. International Semantic Web Conference*, October 2003.

[255] D. Maier. Database Desiderata for an XML Query Language. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[256] A. Malhotra, J. Melton, and N. Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*, 2004, W3C, Working Draft.

[257] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2 2005.

[258] F. Manola, E. Miller, and B. McBride. *RDF Primer*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/rdf-primer/`.

[259] M. Marchiori. The Pseudo Natural Language Interface. Online only, 2004.

[260] M. Marchiori, A. Epifani, and S. Trevisan. Metalog v2.0: Quick User Guide. Technical report, W3C, 2004.

[261] M. Marchiori and J. Saarela. Query + Metadata + Logic = Metalog. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[262] M. Marchiori and J. Saarela. Towards the Semantic Web: Metalog. Online only, 1999.

[263] J. Marsh and D. Orchard. *XML Inclusions (XInclude) Version 1.0*, 2004, W3C, Candidate Recommendation. Available from: `http://www.w3.org/TR/xinclude/`.

[264] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 23–34, 2004.

[265] M. Marx. Conditional XPath, the First Order Complete XPath Dialect. In *Proc. ACM Symposium on Principles of Database Systems*, pages 13–22. ACM, 6 2004.

[266] M. Marx. XPath with Conditional Axis Relations. In *Proc. Extending Database Technology*, 2004.

[267] K. Matsuyama, M. Kraus, K. Kitagawa, and N. Saito. A Path-Based RDF Query Language for CC/PP and UAProf. In *Proc. IEEE Conference on Pervasive Computing and Communications Workshops*, 2004.

[268] N. May, S. Helmer, and G. Moerkotte. Quantifiers in XQuery. In *Proc. Int. Conf. on Web Information Systems Engineering*, 2003.

[269] W. May. LoPiX: a System for XML data Integration and Manipulation. In *Proc. International Conference on Very Large Databases*, 2001.

[270] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 3(4):499–526, 2004.

[271] D. McGuinness and F. van Harmelen. *OWL Web Ontology Language—Overview*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/owl-features/`.

[272] E. Meijer, W. Schulte, and G. Bierman. Programming with Circles, Triangles and Rectangles. In *Proc. XML Conference and Exhibition*, 2003.

[273] E. Meijer and M. Shields. XMLambda: A functional language for constructing and manipulating XML documents. Online only, 1999.

[274] S. Melnik. Simplified Syntax for RDF. Online only, December 1999.

[275] S. Melnik. Representing UML in RDF. Online only, 2000.

[276] H. Meuss and K. U. Schulz. Complete Answer Aggregates for Treelike Databases: a novel Approach to combine querying and navigation. *ACM Transactions on Information Systems*, 19(2):161–215, 2001.

[277] H. Meuss, K. U. Schulz, and F. Bry. Towards Aggregated Answers for Semistructured Data. In *Proc. Int. Conf. on Database Theory*, pages 346–360. Springer-Verlag, 2001.

[278] H. Meuss, K. U. Schulz, F. Weigel, S. Leonardi, and F. Bry. Visual Exploration and Retrieval of XML Document Collections with the Generic System X2. *Journal on Digital Libraries*, 2005.

[279] H. Meyer, I. Bruder, A. Heuer, and G. Weber. The Xircus Search Engine. In *INEX Workshop*, pages 119–124, 2002.

[280] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Proc. ACM Symposium on Principles of Database Systems*, pages 65–76. ACM Press, 2002.

[281] L. Miller. Inkling: RDF query using SquishQL. Online only, 2004.

[282] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *Proc. International Semantic Web Conference*, June 2002.

[283] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 11–22. ACM, 2000.

[284] K. D. Munroe and Y. Papakonstantinou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *Proc. Conf. on Visual Database Systems*, pages 277–296. Kluwer, B.V., 2000.

[285] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML Access Control using Static Analysis. In *Proc. ACM Conf. on Computer and Communications Security*, pages 73–84. ACM Press, 2003.

[286] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: A P2P Networking Infrastructure Based on RDF. In *Proc. International World Wide Web Conference*, May 2002.

[287] M. Nilsson and W. Siberski. RDF Query Exchange Language (QEL): Concepts, Semantics and RDF Syntax. Online only, February 2004.

[288] M. Nilsson, W. Siberski, and J. Tane. Edutella Retrieval Service: Concepts and RDF Syntax. Online only, June 2004.

[289] M. Odersky. Report on the Programming Language Scala. Technical report, Ecole Polytechnique Federale de Lausanne, 2002.

[290] U. Ogbuji. Thinking XML: Basic XML and RDF techniques for knowledge management: Part 6: RDF Query using Versa. Online only, April 2002.

[291] U. Ogbuji. Versa by example. Online only, 2004.

[292] R. Oldakowski and C. Bizer. RAP: RDF API for PHP. In *Proc. International Workshop on Interpreted Languages*, 2004.

[293] B. Oliboni and L. Tanca. A Visual Language should be easy to use: a Step Forward for XML-GL. *Information Systems*, 27(7):459–486, 2002.

[294] F. Olken and J. McCarthy. Requirements and Desiderata for an XML Query Language. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[295] M. Olson and U. Ogbuji. Versa Specification. Online only, 2003.

[296] D. Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/Ph.D. thesis, University of Munich, 1 2005.

[297] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML Data Management*, LNCS 2490. Springer Verlag, 2002.

[298] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *LNCS*. Springer-Verlag, 3 2002.

[299] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML Node Labels. In *Proc. ACM SIGMOD Conf.*, pages 903–908. ACM Press, 2004.

[300] K. Ono, T. Koyanagi, M. Abe, and M. Hori. XSLT Stylesheet Generation by Example with WYSIWYG Editing. In *Proc. Symposium on Applications and the Internet*, 2002.

[301] N. Onose and J. Simeon. XQuery at your Web Service. In *Proc. Int. World Wide Web Conf.*, pages 603–611. ACM Press, 2004.

[302] S. Palmer. Pondering RDF Path. Online only, 2003.

[303] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proc. International Conference on Data Engineering*, pages 251–260, 1995.

[304] P. Patel-Schneider, P. Hayes, and I. Horrocks. *OWL Web Ontology Language—Semantics and Abstract Syntax*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/owl-semantics/`.

[305] P. Patel-Schneider and J. Simeon. The Yin/Yang Web: XML Syntax and RDF Semantics. In *Proc. International World Wide Web Conference*, May 2002.

[306] S. Pepper and G. Moore. *XML Topic Maps (XTM) 1.0*, 2001, TopicMaps.org, Specification. Available from: `http://www.topicmaps.org/xtm/index.html`.

[307] E. Pietriga, J.-Y. Vion-Dury, and V. Quint. VXT: a Visual Approach to XML Transformations. In *Proc. ACM Symposium on Document Engineering*, pages 1–10. ACM Press, 2001.

[308] R. Pinchuk. Toma - Topic Map Query Language. Online only, 2004.

[309] M. Plusch. *Water: Simplified Web Services and XML Programming*. Wiley, 2002.

[310] E. Prud'hommeaux. Algae2 HOTWO. Online only, 2003.

[311] E. Prud'hommeaux. Algae Extension for Rules. Online only, 2004.

[312] E. Prud'hommeaux. Algae RDF Query Language. Online only, 2004.

[313] E. Prud'hommeaux. Optimized RDF Access to Relational Databases. Online only, 2004.

[314] E. Prud'hommeaux and B. Grosof. RDF Query Survey. Online only, 2004.

[315] E. Prud'hommeaux and A. Seaborne. BRQL – A Query Language for RDF. Online only, 2004.

[316] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, February 2005.

[317] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Working draft, W3C, 2 2005.

[318] A. Reggiori and A. Seaborne. RDF Query and Rule languages: Use Cases and Examples. Online only, 2004.

[319] A. Reggiori and D.-W. van Gulik. RDFStore—Perl API for RDF Storage. Online only, 2004.

[320] D. Reynolds. RDF-QBE: a Semantic Web Building Block. Technical Report HPL-2002-327, HP Labs, 2002.

[321] J. Robie. The Syntactic Web: Syntax and Semantics on the Web. In *Proc. XML Conference and Exposition*, December 2001.

[322] J. Robie. Updates in XQuery. In *XML Conference & Exhibiton*, 2001.

[323] J. Robie, E. Derksen, P. Frankhauser, E. Howland, G. Huck, I. Macherius, M. Murata, M. Resnick, and H. SchÂŽning. XQL (XML Query Language). Online only, 1999.

[324] J. Robie, L. M. Garshol, S. Newcomb, M. Fuchs, L. Miller, D. Brickley, V. Christophides, and G. Karvounarakis. The Syntactic Web: Syntax and Semantics on the Web. *Markup Languages: Theory and Practice*, 3(4):411–440, 2001.

[325] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[326] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[327] S. Russell and S. Palmer. Quads. Online only, 2002.

[328] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[329] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, August 2004.

[330] S. Schott and M. L. Noga. Lazy XSL Transformations. In *Proc. ACM Symposium on Document Engineering*, pages 9–18. ACM Press, 2003.

[331] T. Schwentick. XPath Query Containment. *SIGMOD Record*, 2004.

[332] A. Seaborne. RDQL – A Query Language for RDF. Online only, January 2004.

[333] A. Seaborne. RDQL – RDF Data Query Language. Online only, 2004.

[334] A. Seaborne. A Programmer's Introduction to RDQL. Online only, 2002 April.

[335] D. Seipel. Processing XML-Documents in Prolog. In *Workshop on Logic Programming*, 2002.

[336] D. Seipel and J. Baumeister. Declarative Methods for the Evaluation of Ontologies. *KI–KÃijnstliche Intelligenz*, 4:51–57, 2004.

[337] D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management*, 2004.

[338] R. Shearer. REX evaluation. Online only, 2004.

[339] M. Sheshagiri and A. Kunjithapatham. A FIPA Compliant Query Mechanism Using DAML Query Language (DQL). Online only, 2004.

[340] J. E. Simpson. *XPath and XPointer*. O'Reilly, 1st edition, 9 2002.

[341] M. Sintek and S. Decker. TRIPLE—An RDF Query, Inference, and Transformation Language. In *Proc. Deductive Database and Knowledge Management*, October 2001.

[342] M. Sintek and S. Decker. TRIPLE—A Query, Inference, and Transformation Language for the Semantic Web. In *Proc. International Semantic Web Conference*, June 2002.

[343] M. Smith, C. Welty, and D. McGuinness. *OWL Web Ontology Language—Guide*, 2004, W3C, Recommendation. Available from: `http://www.w3.org/TR/owl-guide/`.

[344] A. Souzis. RxML 1.0 Specification. Online only, 2004.

[345] A. Souzis. RxPath. Online only, 2004.

[346] A. Souzis. RxPath Specification Proposal. Online only, 2004.

[347] A. Souzis. RxSLT. Online only, 2004.

[348] A. Souzis. RxUpdate. Online only, 2004.

[349] M. Sperberg-McQueen. How can Tom butter his bread with a knife, if there is a dearth of bread in the larder?, 2001.

[350] D. Steer. TreeHugger 1.0 Introduction. Online only, 2003.

[351] P. Stickler. CBD—Concise Bounded Description. Online only, 2004.

[352] A. Swartz. RDFPath Proposal. Online only, 2001.

[353] I. Tatarinov and A. Halevy. Efficient Query Reformulation in peer Data Management Systems. In *Proc. ACM SIGMOD Conf.*, pages 539–550. ACM Press, 2004.

[354] J. Tennison. *XSLT and XPath On The Edge*. John Wiley, 10 2001.

[355] S. Tessaris. *Questions and Answers: Reasoning and Querying in Description Logic*. PhD thesis, University of Manchester, Department of Computer Science, April 2001.

[356] A. Theobald and G. Weikum. The XXL Search Engine: Ranked Retrieval of XML Data using Indexes and Ontologies. In *Proc. ACM SIGMOD Conf.*, pages 615–615. ACM Press, 2002.

[357] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*, 2001, W3C, Recommendation. Available from: `http://www.w3.org/TR/xmlschema-1/`.

[358] K. Tolle and F. Wleklinski. easy RDF Query Language (eRQL). Online only, 2004. Available from: `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`.

[359] A. Tozawa. Towards Static Type Checking for XSLT. In *Proc. ACM Symposium on Document Engineering*, pages 18–27. ACM Press, 2001.

[360] A. Trombetta and D. Montesi. Equivalences and Optimizations in an Expressive XSLT Fragment. In *Proc. Int. Database Engineering and Applications Symposium*, 2004.

[361] J. Ullman. *Principles of Database and Knowledge-Base Systems*. W. H. Freeman, 1990.

[362] L. Villard and N. LayaÃŕda. An Incremental XSLT Transformation Processor for XML Document Manipulation. In *Proc. Int. World Wide Web Conf.*, pages 474–485. ACM Press, 2002.

[363] P. Wadler. Two semantics for XPath. Online only, 2000.

[364] G. Wagner. Seven Golden Rules for a Web Rule Language. *IEEE Intelligent Systems*, 18(5), 2003.

[365] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation. In *Proc. International Conference on Functional Programming*, 1999.

[366] N. Walsh. RDF Twig: accessing RDF graphs in XSLT. In *Proc. Extreme Markup Languages*, 2003.

[367] J. W. W. Wan and G. Dobbie. Mining Association Rules from XML data using XQuery. In *Proc. Workshop on Australasian Information Security, Data Mining Web Intelligence, and Software Internationalisation*, pages 169–174. Australian Computer Society, Inc., 2004.

[368] S. Waworuntu and J. Bailey. XSLTGen: A System for Automatically Generating XML Transformations via Semantic Mappings. In *Proc. Int. Conf. on Conceptual Modeling*, 2004.

[369] F. Weigel. A Survey of Indexing Techniques for Semistructured Documents. Master's thesis, Institute for Informatics, University of Munich, `http://www.pms.ifi.lmu.de/index.html#PA_Felix.Weigel`, 2002.

[370] N. Wiegand. Investigating XQuery for Querying across Database Object Types. *SIGMOD Record*, 31(2):28–33, 2002.

[371] U. Wiger. XMErl—Interfacing XML and Erlang. In *Proc. International Erlang User Conference*, 2000.

[372] A. Wilk and W. Drabent. On Types for XML Query Language Xcerpt. In *Proc. Workshop on Principles and Practice of Semantic Web Reasoning*, LNCS 2901. Springer-Verlag, 2003.

[373] C. Wilper. RIDIQL Reference. Online only, 2004.

[374] P. T. Wood. On the Equivalence of XML Patterns. In *Proc. Int. Conf. on Computational Logic*, pages 1152–1166. Springer-Verlag, 2000.

[375] C. Zaniolo. The Database Language GEM. In *Proc. ACM SIGMOD Conf.*, 1983.

[376] X. Zhang, K. Dimitrova, L. Wang, M. E. Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. Rainbow: multi-XQuery Optimization using Materialized XML Views. In *Proc. ACM SIGMOD Conf.*, pages 671–671. ACM Press, 2003.

[377] X. Zhang, B. Pielech, and E. A. Rundesnteiner. Honey, I shrunk the XQuery!: an XML Algebra Optimization Approach. In *Proc. International Workshop on Web Information and Data Management*, pages 15–22. ACM Press, 2002.

[378] M. Zoof. Query By Example. In *Proc. AFIPS National Computer Conference*, 1975.

[379] M. Zoof. Query By Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.

Appendix A

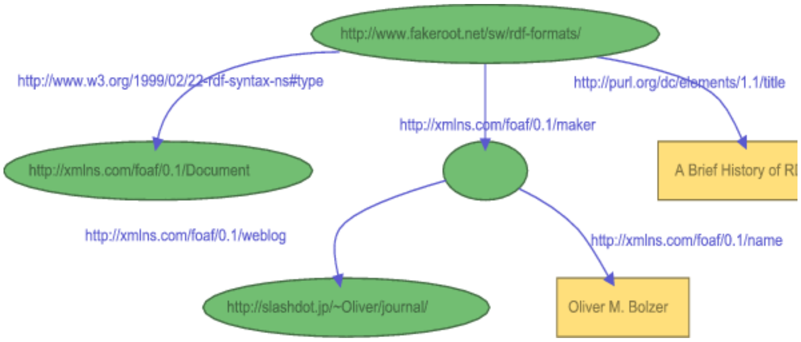
A Brief History of RDF Serialization Formats

A.1 Introduction

The *Resource Description Framework* (RDF) is a language for making simple statements about resources on the World Wide Web in form of a graph of nodes and edges representing the resources, their properties and values. A standard XML syntax for serializing RDF graphs, RDF/XML, exists. Yet, in the five years after the initial release of RDF, numerous alternative serialization formats have been proposed. This report attempts to present an overview of the different proposals and the motivations behind their creation.

The serialization formats presented here can be categorized into three general classes, depending on the use of XML and how a RDF graph is seen: either as fully connected graph or as collection of subject-property-object triples. RDF/XML and it's simplification attempts try to map the nodes and edges of a RDF graph directly to elements in a XML tree. The plain-text formats deriving from N3 concentrate on the individual triples that make up the graph and record them in a non-XML form. The newest generation of serialization formats, TriX and RXR also focus on the triples, but use XML to specify them. A forth class of formats, specially designed for embedding RDF data into XHTML and other XML languages are not considered in this report.

Figure A.1 A sample RDF Graph


http://www.fakeroot.net/sw/rdf-formats/
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://purl.org/dc/elements/1.1/title
http://xmlns.com/foaf/0.1/maker
http://xmlns.com/foaf/0.1/Document
A Brief History of RI
http://xmlns.com/foaf/0.1/weblog
http://xmlns.com/foaf/0.1/name
http://slashdot.jp/~Oliver/journal/
Oliver M. Bolzer

The RDF graph presented in Figure A.1 will be used throughout this report, serialized into each format. It describes this report and it's author using the Friend-Of-A-Friend [75] and Dublin Core [213] vocabularies.

A.2 RDF/XML: The W3C Recommendation

When RDF became a W3C Recommendation for the first time in 1999, [246] defined together with the formal RDF model a XML based syntax for serializing RDF graphs: RDF/XML. Because of the differences in the underlying information models of RDF and XML, one being an edge-and-node-labeled directed graph of resources and properties identified by URIs and the other being a node-labeled tree of elements and attributes identified by the combination of namespace and tag name, the specification proposed a mapping where both resources and properties where converted to XML elements and nested into each other. By using the XML namespace mechanism, it was possible to split an URI into two parts and form an XML element name. The main concept behind RDF/XML later became to be known as *striping* [73], as resources and properties alternate in the nested XML structure.

Query 29. RDF/XML describing this report and it's author.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:foaf="http://xmlns.com/foaf/0.1/" >
<rdf:Description rdf:about="http://www.fakeroot.net/sw/rdf-formats/">
<rdf:type>
<rdf:Description rdf:about="http://xmlns.com/foaf/0.1/Document" />
</rdf:type>
<dc:title>A Brief History of RDF Serialization Formats</dc:title>
<foaf:maker>
<rdf:Description>
<foaf:name>Oliver M. Bolzer</foaf:name>
<foaf:weblog>
<rdf:Description rdf:about="http://slashdot.jp/~Oliver/journal/" />
</foaf:weblog>
</rdf:Description>
</foaf:maker>
</rdf:Description>
</rdf:RDF>
```

Example 29 is a RDF/XML document for the RDF graph from Figure A.1. `rdf:Description` elements represents the resource identified by the URI in it's `rdf:about` attribute. Blank nodes do not have a `rdf:about` attribute, but can optionally have a `rdf:nodeID` attribute to distinguish them from other blank nodes in the same graph. Direct children of a `rdf:Description` elements are the properties describing the resource. For these, the URI identifying the property is used as the element name, by splitting the URI into a prefix and a suffix, used as XML namespace and local part.

RDF/XML allows to shorten the serialization using various abbreviations. For instance, a property with a literal as object can be expressed using a XML attribute. Also the resource that is the object of an statement can be named in the `rdf:resource` attribute of the element for the property, instead of an `rdf:Description` child element. Also the value of a resource's `rdf:type` property can be used as the resouce's element name instead of `rdf:Description`. Example 30 is also a RDF/XML document for Figure A.1, but much shorter then the previous example, through the use of abbreviations.

Query 30. RDF/XML with abbreviations.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
```

```
xmlns:foaf="http://xmlns.com/foaf/0.1/" >
<foaf:Document rdf:about="http://www.fakeroot.net/sw/rdf-formats/"
dc:title="A Brief History of RDF Serialization Formats" >
<foaf:maker>
<rdf:Description foaf:name="Oliver M. Bolzer">
<foaf:weblog rdf:resource="http://slashdot.jp/~Oliver/journal/" />
</rdf:Description>
</foaf:maker>
</rdf:Description>
</rdf:RDF>
```

Many more possibilities to serialize the same graph using RDF/XML exist, because the aggregation of multiple statements about a single resource into children of a single `rdf:Description` element are not mandatory. Also, instead of deeply nesting resources and properties, it is possible to create a shallow structure with all `rdf:Description` elements directly under the root. Example 31 is another serialization of Figure A.1, this time in a very verbose way.

Query 31. Very verbose RDF/XML.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:foaf="http://xmlns.com/foaf/0.1/" >
<rdf:Description rdf:about="http://www.fakeroot.net/sw/rdf-formats/">
<dc:title>A Brief History of RDF Serialization Formats/<dc:title>
</rdf:Description>
<foaf:Document rdf:about="http://www.fakeroot.net/sw/rdf-formats/">
<foaf:maker rdf:nodeID="oliver">
</rdf:Description>
<rdf:Description rdf:nodeID="oliver">
<foaf:name>Oliver M. Bolzer</foaf:name>
</rdf:Description>
<rdf:Description rdf:nodeID="oliver">
<foaf:weblog rdf:resource="http://slashdot.jp/~Oliver/journal/" />
</rdf:Description>
</rdf:RDF>
```

Examples 29-31 all represent the exact same RDF graph. After being processed by a RDF/XML parser, there should be no difference between them. But from the view point of XML tools, each one is a completely different XML document. Because of this syntactic variability, it is very difficult to use standard XML tools like XPath, XSLT and XQuery with arbitrary RDF/XML documents.

Furthermore, several other problems have been identified with RDF/XML, which all led to the development of the subsequent serialization formats. The following are among the most notable:

- It is impossible to distinguish an XML element for a RDF node from one for a property without knowledge about the striping.
- The triples that make up the RDF graph are hard to make out.
- Many different things are used to to represent URIs: element names, attribute names and attribute values.
- It is impossible to specify a single DTD or XML Schema that validates all RDF/XML documents.
- Difficult to read for humans.

In 2001, the RDF Core Working Group was created, partly to fix the RDF/XML syntax and clean up the whole specification. This effort led to the revised RDF/XML Syntax Specification [38], which became a W3C Recommendation in early 2004. However, tough the specification has undergone a major clean up and the syntax is now specified in a cleaner and much more concise manner, the basics have not changed and the problems arising from RDF/XML's structure have not been solved. [36] gives an overview of the problems of the first RDF/XML specification as well as the issues identified during the revision process.

A.3 Simplified Syntaxes for RDF/XML

A.3.1 Unstriped Syntax

Only few month after the publication of [246], Tim Berners-Lee started experimenting with simplifications of RDF/XML. In [47] he considered a modification of RDF/XML without the node/property striping, named "Unstriped Syntax". In it, XML elements are only used for the edges in the RDF graph, with the subjects specified using the newly introduced `rdf:for` attribute.

Query 32. Single Statement using the Unstriped RDF/XML Syntax.

```
<dc:title rdf:for="http://www.fakeroot.net/sw/rdf−formats/">A Brief History of RDF
Serialization Formats</dc:title>
```

Alternately, a default subject for all nested elements can be be given using a `rdf:about` attribute on the parent element, similar to RDF/XML. Blank nodes and deep nesting of elements are handled as in RDF/XML.

Query 33. Figure A.1 serialized using the Unstriped Syntax.

```
<someelement rdf:about="http://www.fakeroot.net/sw/rdf−formats/">
<rdf:type rdf:about="http://xmlns.com/foaf/0.1/Document" />
<dc:title>A Brief History of RDF Serialization Formats</dc:title>
<foaf:maker>
<foaf:weblog rdf:about="http://slashdot.jp/~oliver/journal/" />
<foaf:name>Oliver M. Bolzer</foaf:name>
</foaf:maker>
</someelement>
```

However, lacking a suitable parent element (somelement in the above example), use of the `rdf:Description` element is recommended, undermining the Unstriped Syntax's basic principle that elements are only to be used for properties.

Because it addresses only the striping issue and none of the other problems with RDF/XML, the Unstriped Syntax has not been pursued further than it's "strawman draft" status.

A.3.2 Simplified Syntax

Inspired by the Unstriped Syntax, Sergey Melnik followed up with another simplified RDF/XML syntax [274], in which each element's type, whether it is a node or an edge of the RDF graph, can be determined by looking at it's attributes. As with the Unstriped Syntax, each XML element by default denotes an edge of the graph. The subject is specified in a `rdf:for` attribute and it's object in a `rdf:resource` attribute or in case of a blank node, using child elements. In absence of a `rdf:for` attribute, the subject is defined to be the parent element's object, instead of an explicitly set "default subject", removing the need to use an extra element to denote a resource.

Query 34. Figure A.1 serialized using the Simplified Syntax.

```
<rdf:type rdf:for="http://www.fakeroot.net/sw/rdf−formats/"
rdf:resource="http://xmlns.com/foaf/0.1/Document">
<dc:title rdf:for="http://www.fakeroot.net/sw/rdf−formats/">A Brief History of RDF Serialization
Formats</dc:title>
```

```
<foaf:maker rdf:for="http://www.fakeroot.net/sw/rdf−formats/">
<foaf:name>Oliver M. Bolzer</foaf:name>
<foaf:weblog rdf:resource="http://slashdot.jp/~Oliver/journal/" />
</foaf:maker>
```

Only when an element has a rdf:instance attribute, does it denote a node identified by the URI in the attribute's value. In such a case, the element's name is taken to be the node's rdf:type. Though this feature reintroduces striping, it is explicit and easily detectable. Example 35 shows parts of our example RDF graph utilizing this feature.

Query 35. Simplified Syntax using rdf:instance.

```
<foaf:Document rdf:instance="http://www.fakeroot.net/sw/rdf−formats/">
<dc:title>A Brief History of RDF Serialization Formats</dc:title>
<foaf:maker>...</foaf:maker>
</foaf:Document>
```

A.3.3 XMP

Faced with the need to embed meta-data into the various media formats produced by it's tools, Adobe decided to adopt RDF as the core of it's "Extensible Metadata Platform" [8]. Instead of going for the full RDF/XML format, Adobe uses only a proper subset, disallowing XML literals and reification in order to simplify processing and reduce the complexity of the expressible RDF graphs. As these removed features are rarely used, it is likely that the majority of RDF/XML documents on the Internet are also valid XMP data.

A.3.4 Normalized RDF

In 2001, Jonathan Robie demonstrated that a "normalized" form of RDF/XML could be effectively queried using XQuery, an XML query language without any knowledge about RDF. In [321] he argued, that by standardizing on one of the many possible syntactic variants of RDF/XML, it would be possible to use standard XML tools to effectively query, transform and otherwise process RDF/XML documents.

Going through several refining steps in his paper, Robie arrived at a flat form, where all statements about a single resource are grouped together and all groups put under a common parent, thus avoiding deep nesting of statements.

Query 36. Figure A.1 in "normalized" RDF/XML.

```
<rdf:Description about="http://www.fakeroot.net/sw/rdf−formats/">
<rdf:type>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</rdf:type>
<dc:title>A Brief History of RDF Serialization Formats</dc:title>
<foaf:maker>http://fakeroot.net/staff/Oliver</foaf:maker>
</rdf:Description>
<rdf:Description about="http://fakeroot.net/staff/Oliver">
<foaf:name>Oliver M. Bolzer</foaf:name>
<foaf:weblog>http://slashdot.jp/~Oliver/journal/</foaf:weblog>
</rdf:Description>
```

Being only a technical demonstration for the possibility of querying RDF using XQuery, details like the difference between resources and literals as objects or blank nodes are not addressed. In Example 36 a new URI had to be assigned to identify the maker of the document.

A.3.5 RxML

RxML [344] is a serialization format by Adam Souzis created as component of his Rx4RDF suite of RDF-related technologies. It is unique in it's consistent use of XML element names to encode all URIs extending the way properties are handled in RDF/XML to subjects and objects. Each child of the root `rx:rx` element specifies a resource describe in the RxML document. It's children are in turn the properties and the grandchildren are objects: either text children for literals or empty elements for resources. Nesting is not allowed, limiting the maximum depth of the XML tree to 3. Blank nodes are identified by resources who's URIs begin with 'bnode:'.

Query 37. Figure A.1 in RxML

```
<rx:rx xmlns:rx='http://rx4rdf.sf.net/ns/rxml#'
        xmlns:bnode='bnode:'
        xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:fakeroot='http://www.fakeroot.net/sw/'
        xmlns:dc='http://purl.org/dc/elements/1.1/'
        xmlns:foaf='http://xmlns.com/foaf/0.1/' >
  <fakeroot:rdf-formats>
    <rdf:type><foaf:Document /></rdf:type>
    <dc:title>A Brief History of RDF Serialization Formats</dc:title>
    <foaf:maker><bnode:Oliver /></foaf:maker>
  </fakeroot:rdf-formats>
  <bnode:Oliver>
    <foaf:name>Oliver M. Bolzer</foaf:name>
    <foaf:weblog><journal xmlns="http://slashdot.jp/~Oliver/" /></foaf:weblog>
  </bnode:Oliver>
</rx:rx>
```

While the consistent use of XML element names for URIs seems an elegant solution, it is accompanied by a fatal problem: some URIs can't be expressed due to restrictions in the characters allowed for element names in XML. In Example 37, the document's URI (*http://www.fakeroot.net/sw/rdf-formats/*) can't be turned into a XML element name because of the trailing '/' that had to be omitted.

A.4 Plain-Text Formats

A.4.1 Notation 3

Meanwhile, giving up on a usable XML syntax for RDF, Tim Berners-Lee proposed Notation 3, also known as N3 [49]. Contrary to previous serialization formats, N3 is not a XML based format. Born out of a pseudo-syntax people started using in various discussion forums instead of RDF/XML, N3 focuses on the triples that make up a RDF graph and writes them down in a straight manner: subject, property, object .

Query 38. A single Statement in N3.

```
<http://www.fakeroot.net/sw/rdf-formats/> <http://purl.org/dc/elements/1.1/title>
"A Brief History of RDF Serialization Formats" .
```

Two shortcuts are provided to combine several statements. A semicolon introduces another property about the same subject and a comma introduces another object with the same property and subject. Blank nodes are identified by using square brackets as objects, putting the statements about that blank node inside the brackets. Additionally, N3 allows the use of short prefixes to abbreviate long URIs, similar to the namespace prefixing mechanism in XML. Also the very common rdf:type predicate can be abreviated to just an a. [50] gives an excellent introduction to the basics of N3.

Query 39. Figure A.1 in N3.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .
<http://www.fakeroot.net/sw/rdf-formats/> a foaf:Document;
dc:title "A Brief History of RDF Serialization ...";
foaf:maker [
foaf:weblog <http://slashdot.jp/~Oliver/journal>;
foaf:name "Oliver M. Bolzer" ] .
```

N3 supports reification by quoting statements within curly braces. A quoted statement can then be used as subject for another statement.

Query 40. Reification in N3.

```
{ ex:Moby_Dick foaf:maker ex:Oliver .} ex:trustable ex:false .
```

N3 is also not just an serialization format for RDF graphs. It has additional support beyond RDF, allowing whole graphs to be quoted as well as formulation of rules and queries using variables and quantification. While most people only think of N3 as a serialization format, some people think of N3 as a rule language, while others consider it a query language. To avoid confusion, attempts have been made to define subsets of N3 according to capability: N3 RDF, N3 Rules and full N3.

Being easy to read for both humans and machines, N3 was quickly adopted by the Semantic Web community as the format used for online discussions about RDF. Today, various tools and query language implementations for RDF accept N3 as input and output format together with RDF/XML.

A.4.2 N-Triples

N-Triples is a minimalist subset of N3, only allowing one triple per line without any abbreviations. Designed for RDF Test Cases [186], it is intended to be extremely easy to parse and generate by scripts. To avoiding any nesting, Blank nodes need to be identified by a temporary identifier starting with _:. N-Triples neither supports URI abbreviation, nor reification.

Query 41. Figure A.1 in N-Triples.

```
<http://www.fakeroot.net/sw/rdf-formats/> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.org/foaf/0.1/Document> .
<http://www.fakeroot.net/sw/rdf-formats/> <http://purl.org/dc/elements/1.1/title>
"A Brief History of RDF Serialization Formats" .
<http://www.fakeroot.net/sw/rdf-formats/> <http://xmlns.org/foaf/0.1/maker> _:a .
_:a <http://xmlns.org/foaf/0.1/weblog> <http://slashdot.jp/~Oliver/journal> .
_:a <http://xmlns.org/foaf/0.1/name> "Oliver M. Bolzer" .
```

Because of it's precise and simple syntax and straight accordance with the RDF's concept of triples ([233]), N-Triples is often encountered in introductory documents about RDF, such as [258].

A.4.3 Quads

When aggregating RDF statements from multiple sources and saving them locally, tracking the origin, or context, of each statement becomes more and more important. Some storage systems store the URI of origin together with each triple, forming a "quad". Quads [327] is an extention of N-Triples, adding an optional forth element for such context information.

Query 42. A single statement that originated at *http://www.fakeroot.net/sw/SampleG.rdf*, in Quads

```
<http://www.fakeroot.net/sw/rdf-formats> <http://purl.org/dc/elements/1.1/title>
"A Brief History of RDF ..." <http://www.fakeroot.net/sw/SampleG.rdf> .
```

The specification of Quads includes further extentions such as "compound names" and statement terminators other than the dot, but the semantics of them are not further explained.

A.4.4 Turtle

While more and more RDF-related tools adopted N3 in addition to RDF/XML, most of them implemented only a ad-hoc subset of N3, leaving out some of the more complex features that go beyond the RDF model. In light of such development, Dave Beckett proposed in the end of 2003 a new plain-text format Turtle ([20], [37]), extending N-Triples with some of the commonly used and well understood features of N3, while staying within the RDF model. Turtle deliberately skips support for reification.

Among the features taken from N3 are short prefixes for long URIs and the abbreviations using commas and semicolons, as well as blank node creation using square brackets and collections. Also the default character encoding has been changed from US-ASCII to UTF-8.

Query 43. Figure A.1 in Turtle.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .
<http://www.fakeroot.net/sw/rdf-formats/> a foaf:Document;
dc:title "A Brief History of RDF Serialization ...";
foaf:maker [
foaf:weblog <http://slashdot.jp/~Oliver/journal>;
foaf:name "Oliver M. Bolzer" ] .
```

Example 43 is identical to Example 39 given above for N3 above, due to Turtle being mostly a subset of N3.

A.4.5 TriG

TriG [55] is the newest in the plain-text line of formats descending from N3, proposed by Chris Bizer. Dubbed as "a compact and readable alternative to the XML-based TriX" [55], TriG extends Turtle beyond theb RDF model by adding support for serializing multiple graphs in one file, with the ability to give each a distinct name [101].

Query 44. Figure A.1 with additional name in TriG.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.org/foaf/0.1/> .
@prefix fakeroot: <http://www.fakeroot.net/sw/> .
fakeroot:SampleG { fakeroot:rdf-formats/ a foaf:Document;
dc:title "A Brief History of RDF Serialization ...";
foaf:maker [
foaf:weblog <http://slashdot.jp/~Oliver/journal>;
foaf:name "Oliver M. Bolzer" ] .
}
```

Example 44 shows the graph from Figure A.1, given the name *http://www.fakeroot.net/sw/SampleG* . It's triples are grouped together using curly braces, with the name prepended.

A.5 Triple-based XML Formats

During 2003, while completing the revision of RDF/XML, Dave Beckett summarized the inherent problems of RDF/XML and collected requirements and ideas for new serializations formats in [36]. Based on the experiences with N3 and RPV (see below), Beckett argued, that such a formats should be closely based on the RDF graph via the terminology in [233] and be minimal in the number of alternate forms for the same RDF graph.

Though not a format proposal, [36] defines the up-to-date most extensive list of requirements for a new XML format.

A.5.1 RPV

In 2002, despite the increasing popularity of N3, Tim Bray still felt the need for a XML based format and created RPV [68] in order to leverage XML's diverse assets, such as support for Internationalization and widely-deployed base of software. The goals was to create a format that was entirely unambiguous and highly human-readable, by emphasizing the triples that make up a RDF graph.

RPV takes a subject centric view on triples, collecting statements about a single resource together, similar to the Normalized RDF approach by Robie. Instead of using property names as element names, RPV utilizes only two tags: `R` (as in resource) and `PV` (as in property/value). URIs are specified using the attributes `r`, `p` and `v` for resource, property and value, respectively. Blank nodes are emulated by giving a `R` element an `id` attribute but no `r` attribute.

Query 45. Figure A.1 in RPV.

```
<R r="http://www.fakeroot.net/sw/rdf−formats/">
<PV p="http://www.w3.org/1999/02/22−rdf−syntax−ns#type" v="http://xmlns.org/foaf/0.1/Document" />
<PV p="http://purl.org/dc/elements/1.1/title">A Brief History of RDF ...</PV>
<PV p="http://xmlns.org/foaf/0.1/maker" v="#oliver" />
</R>
<R id="oliver">
<PV p="http://xmlns.org/foaf/0.1/weblog" v="http://slashdot.jp/~Oliver/journal" />
<PV p="http://xmlns.org/foaf/0.1/name">Oliver M. Bolzer</PV>
</R>
```

In order to abbreviate long URIs, RPV allows the use of the attributes `rBase`, `pBase`, and `vBase`, providing base URIs similar to `xml:base` for resource, property and value, respectively. These bases apply to the element and all contained elements. As only one such base is allowed for each type, the usability is very limited in situations using vocabularies with varying prefixes.

Query 46. RPV with abbreviated URIs.

```
<R r="http://www.fakeroot.net/sw/rdf−formats/"
pBase="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
vBase="http://xmlns.org/foaf/0.1/" >
<PV p="type" v="Document" />
<PV p="http://purl.org/dc/elements/1.1/title">A Brief History of RDF ...</PV>
```

```
<PV p="http://xmlns.org/foaf/0.1/maker" v="#oliver" />
</R>
```

RPV supports reification using the `rpv` attribute on a `R` element, pointing to another `R` element's id. In Example 46, in the first `R` element, identified as *foo* by the `id` attribute, statements are made about *Mobuy_Dick*. Then in the second `R` element, pointing to the first `R` element using the `rpv` attribute, the statements are stated to be not trustable.

Query 47. Reification in RPV.

```
<R id="foo" r="http://example.org/Moby_Dick">
<PV p="http://xmlns.com/foaf/0.1/maker" v="http://example.org/Oliver" />
</R>
<R rpv="#foo">
<PV p="http://example.org/trustable" v="http://example.org/false" />
</R>
```

Though never actually used in any implementation, RPV animated others to pursue the goal of a XML-based format that emphasizes the triples instead of trying to somehow map RDF graphs to XML trees.

A.5.2 TriX

Following up on the requirements proposed by Beckett, Jeremy J. Carroll and Patrick Stickler in [102] designed the XML-based format TriX. The authors take an unique approach by first defining an absolutely minimal base format without any abbreviations and then using XSLT for syntactic extentions, together with stylesheets that convert to the base format. In addition, TriX goes beyond the original RDF model by supporting Named Graphs [101] and literals as subjects.

A TriX document contains one or more graphs, each optionally with a name. Each graph consists of one or more triples. The `triple` element is the core of TriX, containing three children. The position of each child determines whether the child is the subject, the property or the object of the triple. The element used identifies it's type.The `uri` element is used for unabbreviated URIs, while the `id` element is used for identifying blank nodes. `plainLiteral` is used for String literals, while `typedLiteral` is used for any other type of literal in combination with a `datatype` attribute.

Query 48. Figure A.1 in TriX.

```
<TriX xmlns="http://www.w3.org/2004/03/trix/trix−1/">
<graph>
<uri>http://www.fakeroot.net/sw/SampleG</uri>
<triple>
<uri>http://www.fakeroot.net/sw/rdf-formats/</uri>
<uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
<uri>http://xmlns.org/foaf/0.1/Document</uri>
</triple>
<triple>
<uri>http://www.fakeroot.net/sw/rdf-formats/</uri>
<uri>http://purl.org/dc/elements/1.1/title</uri>
<plainLiteral>A Brief History of RDF Serialization Formats</plainLiteral>
</triple>
<triple>
<uri>http://www.fakeroot.net/sw/rdf-formats/</uri>
<uri>http://xmlns.org/foaf/0.1/maker</uri>
<id>x</id>
</triple>
<triple>
<id>x</id>
<uri>http://xmlns.org/foaf/0.1/weblog</uri>
```

```
<uri>http://slashdot.jp/~Oliver/journal/</uri>
</triple>
<triple>
<id>x</id>
<uri>http://xmlns.org/foaf/0.1/name</uri>
<plainLiteral>Oliver M. Bolzer</plainLiteral>
</triple>
</graph>
</TriX>
```

Example 48 is the sample RDF graph serialized using the basic TriX format. The name *http://www.fakeroot.net/sw/SampleG* is attached to it via the uri element directly under graph. Though being very verbose, the triples are clearly identifiable.

TriX allows syntactic extentions that make the syntax more human-friendly through the use of XSLT. One popular trick to increase readability of RDF serializations is to allow a XML QName-like abbreviation for URIs. By declaring an appropriate stylesheet processing instruction, TriX allows such syntactic sugar. Example 49 is one triple from the graph serialized in TriX, using the `qname` element to abbreviate long URIs.

Query 49. Syntactic Extentions in TriX using XSLT.

```
<?xml-stylesheet type="text/xml"
href="http://www.w3.org/2004/03/trix/all.xsl" ?>
<TriX xmlns="http://www.w3.org/2004/03/trix/trix−1/"
xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xmlns:foaf="http://xmlns.org/foaf/0.1/" >
<graph>
<triple>
<uri>http://www.fakeroot.net/sw/rdf-formats/</uri>
<qname>rdf:type</qname>
<qname>foaf:Document</qname>
</triple>
</graph>
</TriX>
```

Other syntactic sugars demonstrated by the authors of TriX include the use of `xml:base` as another method for URI-abbreviation, tags for specific typed literals and collections. The authors even go as far as suggesting RDF/XML as an TriX extention, based on the possibility of writing an RDF/XML parser in XSLT.

A.5.3 RXR

Discontent with TriX's decision to support features beyond the original RDF model, it's dependency on XSLT and the risk of ad-hoc extentions, Dave Beckett continued with the work he had began in [36] and formulated in [21] another proposal for a triple-centric XML-based format, RXR (Regular XML RDF).

Similar to TriX, the `triple` element, containing three children, is at the heart of RXR. But, instead of relying on the position, the children's roles are unambiguously identified by the elements `subject`, `predicate` and `object`. URIs are then given as value to the `uri` attribute, while literals are given as element content, with an optional `datatype` attribute. Blank nodes are specified with the `blank` attribute.

Query 50. Figure A.1 in RXR.

```
<graph xmlns="http://ilrt.org/discovery/2004/03/rxr/">
<triple>
<subject uri="http://www.fakeroot.net/sw/rdf−formats/" />
<predicate uri="http://www.w3.org/1999/02/22−rdf−syntax−ns#type" />
<object uri="http://xmlns.org/foaf/0.1/Document" />
```

```
</triple>
<triple>
<subject uri="http://www.fakeroot.net/sw/rdf−formats/" />
<predicate uri="http://purl.org/dc/elements/1.1/title />
<object>A Brief History of RDF Serialization Formats</object>
</triple>
<triple>
<subject uri="http://www.fakeroot.net/sw/rdf-formats/" />
<predicate uri="http://xmlns.org/foaf/0.1/maker" />
<object blank="x" />
</triple>
<triple>
<subject blank="x" />
<predicate uri="http://xmlns.org/foaf/0.1/weblog" />
<object uri="http://slashdot.jp/~Oliver/journal/" />
</triple>
<triple>
<subject blank="x" />
<predicate uri="http://xmlns.org/foaf/0.1/name" />
<object>Oliver M. Bolzer</object>
</triple>
</graph>
```

Example 50 shows again the RDF graph from Figure 1, this time serialized using RXR. Despite it's verbosity, the triples are clearly recognizable.

RXR does not allow any abbreviations of URIs or other complexities such as XML literals. One notable exception are collections, supported by RXR through the `collection` element. Multiple statements with the same subject and predicate can be aggregated using this facility.

Query 51. Collections in RXR.

```
<graph xmlns="http://ilrt.org/discovery/2004/03/rxr/">
<triple>
<subject uri="http://example.org/box" />
<predicate uri="http://example.org/contains" />
<collection>
<object>apple</object>
<object>pear</object>
<object>potato</object>
</collection>
</triple>
</graph>
```

Three tripled are contained in Example 51. The same triples could also have been written separately, using three triple elements.

A.6 Features Overview

Feature\Format	RDF/XML	XMP	N3	N-Triples	Turtle	TriG	RPV	TriX basic	TriX ext.	RXR
XML	X	X					X	X	X	X
URI abbreviation	X	X	X		X	X	X		X	
Statement Aggregation	X	X	X		X	X	X			
Deep Nesting	X	X	X		X	X				
Blank Nodes	X	X	X	X	X	X	X	X	X	X
Collections	X	X	X		X	X			X	X
Typed Literals	X	X	X	X	X	X		X	X	X
Reification	X		X			X*	X	X*	X*	
Features beyond RDF			X			X		X	X	

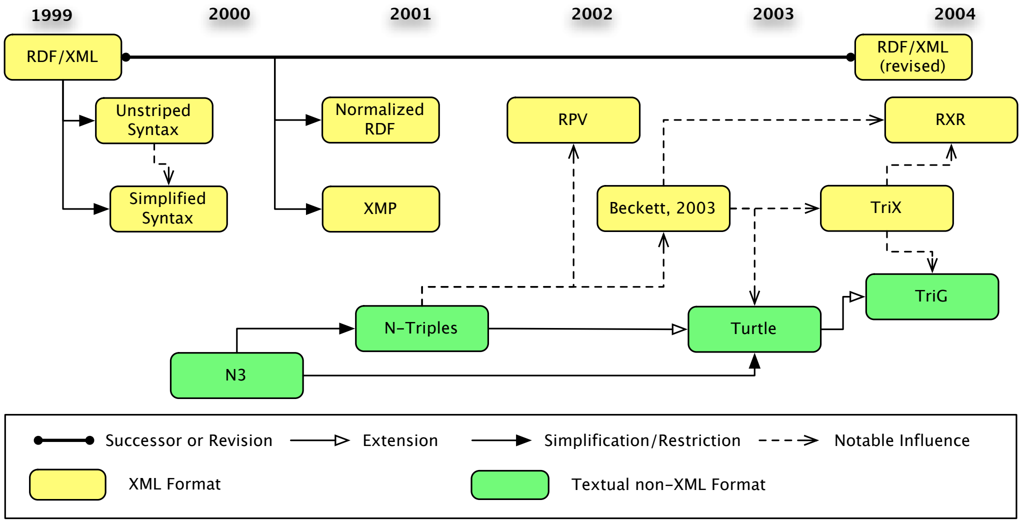
*: Reification using Named Graphs

Normalized RDF, the Unstriped Syntax and Simplified Syntaxe are not listed, as they were only incomplete sketches and not concrete format proposals.

A.7 Genealogy

Figure A.2 is an attempt at portraying the genealogy of the formats described in this document. The newer formats were obviously influenced by most if not all preceding formats. Yet some had a stronger influence, others less. The depicted affinities are based on citations by the proposals and statements made by the respective authors.

Figure A.2 Genealogy of RDF serialization formats
("Beckett, 2003" refers to [36]. Though not a concrete format proposal itself, it is included here as a major source of inspiration and guidance for the subsequent format proposals.)



A.8 Conclusions

The challenge of serializing RDF graphs and the dissatisfaction of the Semantic Web community with RDF/XML has brought forward numerous proposals for alterternate serialization formats, most of which have been described here. After various early attempts to simplify RDF/XML failed to gain support, the idea of directly mapping RDF nodes and edges to XML elements appears to have been abandoned in favor of a more triple-centric view of RDF graphs.

N3 has seen wide adoption by the Semantic Web community as a triple-centric and human-friendly format. However actual implementations vary greatly in the supported features of N3. Current developments indicate a high chance that future implementations will standardize on Turtle as an adequate common denominator of the N3-based proposals.

Still, many in the community feel the need for a triple-centric XML-based format, in order to facilitate interchange between heterogenous systems leveraging existing XML tools. TriX and RXR are the current contestants for such a format, but it is still too early to speculate on which will prevail.

Considering the disputes concerning support for new features like graph naming and literals as subjects, it is likely that the world will see yet more format proposals in the near future. Until some consensus is reached, RDF/XML remains the only formally standartized format all implementations must support.

Appendix B

Evaluation Tables

For space reasons, the detailed evaluation results are given for only a limited selection of languages. The full details can be obtained from the authors and are available on the working group page: `http://rewerse.net/I4`.

Appendix C

Revision History[1]

The content of this deliverable has been revised several times since its publication in August 2004. As part of this revision process additional authors have contributed to the survey, e.g., with a more in-depth look at XSLT, with a section on query languages and rule interchange at the W3C, and with a complete revision of the RDF part focusing more on language constructs and evaluation methods.

The first major revision of the presented material orginiates from a course on Web and Semantic Web query languages for the REWERSE "Reasoning Web" summer school 2005[2]. For this summer school the material of the deliverable has been revised to give the surveyed languaguages a more uniform treatment and to clarify the main aspects of query language research in each area. The material has been published in an LNCS tutorial volume [24].

For the XML Tage 2006[3] a tutorial based on the content of this survey is under preparation. It will focus on the different forms of Web query languages with particular emphasis on W3C recommendations. The material has been updated from the 2005 "Reasoning Web" summer school and a look at the relation between query languages and recent standardization activities on rule languages for the Web has been added.

Finally, a revision focusing on the RDF part of the material is underway for the Reasoning Web summer school 2006[4]. This revision focuses only on RDF query languages, but emphasizes in that part common or innovative language constructs and evaluation methods.

Overall these revisions have contributed to **(a)** disseminate the content of the deliverable D1 to a wider audiences and **(b)** to refine the survey in particular areas, most notably RDF query languages and rule-based query languages.

[1] Major revisions from deliverable I4-D1.

[2] `http://reasoningweb.org/2005/`

[3] `http://www.xmltage.de/`

[4] `http://reasoningweb.org/2006/`