Web and Semantic Web Query Languages: A Survey

James Bailey[1], François Bry[2], Tim Furche[2], and Sebastian Schaffert[2]

[1] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Victoria 3010, Australia
http://www.cs.mu.oz.au/~jbailey/
[2] Institute for Informatics,University of Munich,
Oettingenstraße 67, 80538 München, Germany
http://pms.ifi.lmu.de/

Abstract. 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.

1 Introduction

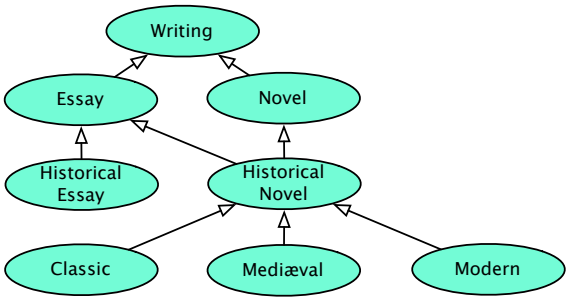
The Semantic Web Vision

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 [37]. 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 Berners-Lee et al. [37]:

> "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."

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:

Fig. 1 A categorisation of books as it might occur in a Semantic Web ontology



> "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." [37]

A number of formalisms have been proposed for representing Semantic Web meta-data, in particular RDF [217], Topic Maps [155], and OWL (formerly known as DAML+OIL) [23, 151]. 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 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 [51] 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 [36]), rules expressed in formalisms like SWRL [150] or RuleML [43] 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?

Structure. After briefly discussing the three different representation formats XML, RDF, and Topic Maps in Section 2.1, each of the languages is introduced with sample queries against a common Semantic Web scenario (cf. Section 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:

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 [206], CDuce [27], XDuce [152], Xtatic (`http://www.cis.upenn.edu/~bcpierce/xtatic/`), Scriptol (`http://www.scriptol.com/`), and Cω (`http://research.microsoft.com/Comega/` [205]). XML APIs are not considered, e.g.: DOM [9], SAX (`http://www.saxproject.org/`), and XmlPull (`http://www.xmlpull.org/`). XML-related language extensions are not considered, e.g.: HaXML [276] for Haskell, XMerL [282] for Erlang, CLP(Flex) [88] for Prolog, or XJ [145] for Java. General-purpose programming languages with Web service support are also not considered, e.g.: XL [115, 116], Scala [218], Water [235].
- *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. [4] 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 [43] 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 [113], 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. [280], for RDF cf. to [190]).

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.

2 Preliminaries

2.1 Three Data Formats: XML, RDF and Topic Maps

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."[3]

An "XML document" is a file, or collection of files, that adheres to the general syntax specified in the XML Recommendation [48], 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.

[3] `http://www.w3.org/XML/`

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[4], and ordered[5] tree, if one does not consider the various referencing or linking mechanisms of XML. Although this interpretation is that of the data model retained for XML (cf. XML Infoset [94], XQuery, XPath [111]) 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.

RDF and RDFS. *RDF* [25, 172] 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 [51, 148, 172, 194], e.g. `rdf:type` for specifying the type of properties, `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 `rdfs:Property`, the class of all properties.[6]

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.

[4] i.e. the number of children of an element is not bounded.
[5] i.e. the children of an element are ordered.
[6] this survey tries to use self-explanatory prefixes for namespaces where possible.

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. [56] for a detailed survey.

OWL [23, 204, 261] extends RDFS with a means for defining description vocabularies for Web resources. OWL is only considered superficially in this survey, cf. Section 1.

Topic Maps. Topic Maps [155, 232] 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 [155], or only by "occurrence types" in the formalism XTM [232].

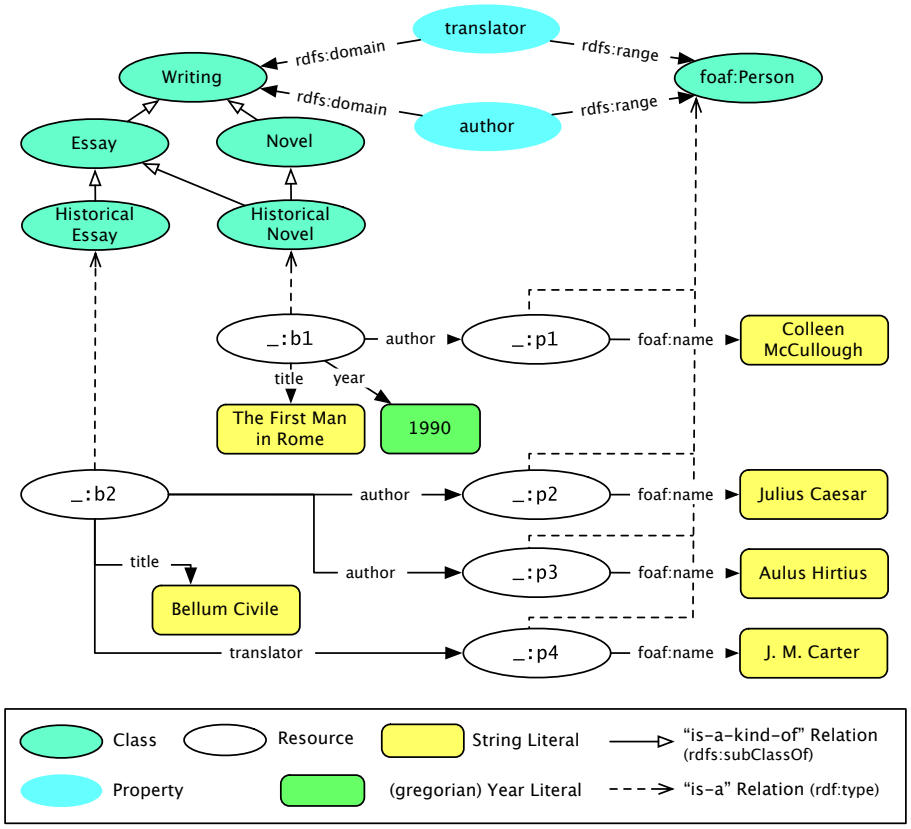
"Topic characteristics" denote the names a topic has, what associations it partakes in, and what its occurrences are. "Facets" (a concept of HyTM but not of XTM) are attribute-value pairs that can be attached to any kind of topic map component for explanation purposes. Facets are thus a means to attach 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 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 [126, 177]. Interestingly, Topic Maps associations are similar to the "extended links" of the XML linking language XLink (`http://www.w3.org/XML/Linking/`).

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 2 depicts this data as a (simplified)

Fig. 2 Sample Data: representation as a (simplified) RDF graph.



RDF graph [51, 172, 184]. 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 [39] 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 [105] 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`.

Sample Data in RDF. The RDF representation of the book recommender system directly corresponds to the simplified RDF graph in Fig. 2. It is given here in the *Turtle* serialisation [24].

```
@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 .
_: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 "_:".

Sample Data in Topic Maps. The Topic Map representation of the book recommender system makes use of the Linear Topic Maps syntax [125]. Subclass-superclass associations are identified using the subject identifiers of XTM [232]. 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.

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.

2.3 Sample Queries

The different query languages are illustrated using five types of queries against the sample data. This categorisation is inspired by Maier [192] and Clark [87].

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 1 *"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 2 *"Select all data items with any relation to the book titled 'Bellum Civile'."*

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 3 *"Select all data items except ontology information and translators."*

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 4 *"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*".

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 5 uses the `max(·)` value aggregation, while Query 6 uses structural aggregation:

Query 5 *"Return the last year in which an author with name 'Julius Caesar' published something."*

Query 6 *"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.

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 7 *"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 8 *"Return the transitive closure of the `subClassOf` relation."*

Not all inference queries are combination queries, as the following example illustrates:

Query 9 *"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.

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 [129, 130] and LGQ [224]. 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 [21] which are not tailored towards querying by humans. Finally, this article does not consider XML query languages focused on information retrieval, e.g., XirQL [120], EquiX [89], ELIXIR [82], XQuery/IR [49], XXL [270], XirCL [210], XRANK [142], PIX [7], XSEarch [90], FleXPath [8], and TeXQuery [6]. 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.

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 [286], an extension of QUEL, and OQL [73]) 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 [286]: "`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 [83, 119], 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 [3] 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)" [127, 230], 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 1 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. 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 [114, 118] 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.

Data Selection with XPath XPath is presented in [86] and [258, 269], 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 [94] 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 [31], a revision of XPath currently under development at the W3C.

Reverse navigation has been considered for generalized path expressions, cf. [68, 69]). However, it has been shown in [225] 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. [199, 200], 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 [108], in XSLT [85], in XPointer [135], in XQuery [42], 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 [128, 225, 275]. 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 [131, 132]. Various sub-languages of XPath (e.g., forward XPath [225], Core or Navigational XPath [131], [26]) and extensions (e.g., CXPath [199]) 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 [149]. 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., [101, 211, 250, 285]. Several methods providing efficient implementations of XPath relying on standard relational database systems have been published, cf., e.g., [137, 138, 226].

Currently, the W3C is, as part of its activity on specifying the XML query language XQuery, developing a revision of XPath: XPath 2.0 [31]. See [164] 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 [193], and (5) a complete formal semantics [104].

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 [203] is syntactically an extension of XPath but its semantics and evaluation are based on logic programming, more specifically F-Logic and FLORID [188]. XPathLog extends the syntax of XPath as follows: (1) variables may occur in path expressions, e.g., `//book[name` $\rightarrow$ `N]` $\rightarrow$ `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 [254] 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 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 [256] and querying OWL ontologies [255].

Project page:
`http://www-info1.informatik.uni-wuerzburg.de/database/research_seipel.html`
Implementation:
not publicly available
Online demonstration:
none

The Transformation Language XSLT XSLT [85], the E**x**tensible **S**tylesheet **L**anguage, 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. [279]), 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 [169] 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 [167]. As with XQuery 1.0, this language is based upon XPath 2.0 [31]. 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 [193] 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. [165].

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 [85]).

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.., [165, 269]).

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 [13], here only some core results are outlined: Formal semantics for (fragments of) XSLT have been investigated in [38, 171]. [169] gives a proof showing that XSLT is Turing complete. Analysis of XSLT is examined in [103], 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 [214] presents a theoretical model of XSLT and examines a number of decision questions for fragments of this model. Work in [198] examines the question of whether the output of an XML transformation conforms to a particular document type. Type checking is also addressed in [272].

Efficient evaluation of XSLT programs is also an important topic. In [156, 186], translations to SQL are considered. Work in [274] describes incremental methods for processing multiple transformations. Work in [249] proposes a lazy

evaluation for XSLT programs, while [166] describes optimizations based on experiences from the widely used XSLT processor Saxon. Other specific techniques for optimizing XSLT programs and evaluation are described in [102, 134, 143, 273]. Further engineering aspects of XSLT programs have also received attention, namely transformation debugging [12] and automatic stylesheet generation [227, 279].

Project page:
`http://www.w3.org/Style/XSL/`
Implementation:
very numerous, see project page
Online demonstration:
none

Fxt. fxt [32], 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. [33]). *Fxt*'s computation model is also more restricted than that of XSLT due to the lack of named templates.

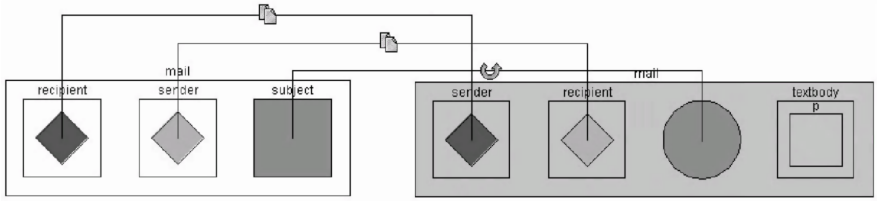
Project page:
`http://atseidl2.informatik.tu-muenchen.de/~berlea/Fxt/`
Implementation:
available from the project page
Online demonstration:
none

VXT. VXT [233] is a visual language and interactive environment for specifying transformations of XML documents. It is based on the general purpose transformation language Circus[7]: Whereas most other XML query languages employ some form of graph-shaped visualization for both data and queries, VXT uses treemaps [157] 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. 3

Project page:
none
Implementation:
not publicly available
Online demonstration:
none

[7] `http://www.xrce.xerox.com/solutions/circus.html`

Fig. 3 Treemap representation of a VXT rule
(Pietriga et al. [233], © ACM Press)



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 [76, 77, 192]. A number of proposals, e.g., XQL and Quilt, have been published in answer to this activity, each with varying influence on XQuery [42], the language currently under standardisation at the W3C:

XQL [244, 246] 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 [79] 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 [42] 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 [104]. 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. [168]).

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. [225], 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 [104]. Comprehensive but easy to follow introductions to XQuery are given in, e.g., [53, 163].

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 [76] 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., [106, 109]). Even before the development has finished, initial practical introductions to XQuery have been published, e.g., [53, 163]. Industry interest is also visible in the simultaneous development of standardized XQuery APIs, e.g., for Java [107], and numerous implementations, both open source (e.g., Galax [112]) and commercial (BEA [117], IPSI-XQ [110]). Aside from these main-memory implementations, one can also find streamed implementations of XQuery (e.g., [22, 173]) where the data flows by as the query is evaluated. First results on implementing XQuery on top of standard relational databases (e.g., [97, 139]) 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"[8].

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 [169]. 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 [80, 81, 100, 202,

[8] http://www-rocq.inria.fr/gemo/Gemo/Projects/XIME-P/

268, 287, 288]. 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 [278], for web service implementation [228], for querying heterogeneous *relational* databases [281], for access control and policy descriptions [216], for synopsis generation [92], and as the foundation of a visual XML query language (XQBE) [10], of a XML query language with full-text capabilities [5, 6], and of an update [54, 78, 243] and reactive [46] language for XML.

Project page:
`http://www.w3.org/XML/Query`
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`

3.2 Research Prototypes: The Positional Approach to XML Querying

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. [290]), 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).

UnQL. UnQL [64–66] (the *Unstructured Query Language*) is a query language originally developed for querying semistructured data and nested relational databases 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. [65] and a more approachable surface syntax.[9]

The following expression uses functional style pattern matching for selecting all books in a tree.

```
fun f1(T1 ∪ T2)   = f1(T1) ∪ 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

```
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.

[9] The syntax from [64, 65] is used and not the slightly differing syntax in [66].

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 [64, 66]. UnQL's evaluation is founded in graph simulation, see [66]. [64] 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`[10]
Implementation:
available from the project page
Online demonstration:
none

XML-QL. XML-QL [98, 99] 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 [98]) 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:

```
WHERE
  <bookdata>
    <book>
    </> ELEMENT_AS $b
  </>
CONSTRUCT
  <results>
    <result>
      $b
      WHERE <author>
              <name> $n </>
            </> ELEMENT_AS $a
      CONSTRUCT $a
                $n
    </>
  </>
```

[10] Not accessible at the time of writing.

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

XMAS. XMAS [189], the *XML Matching And Structuring language* is an XML query language developed as part of MIX [18] 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}
  </>
```

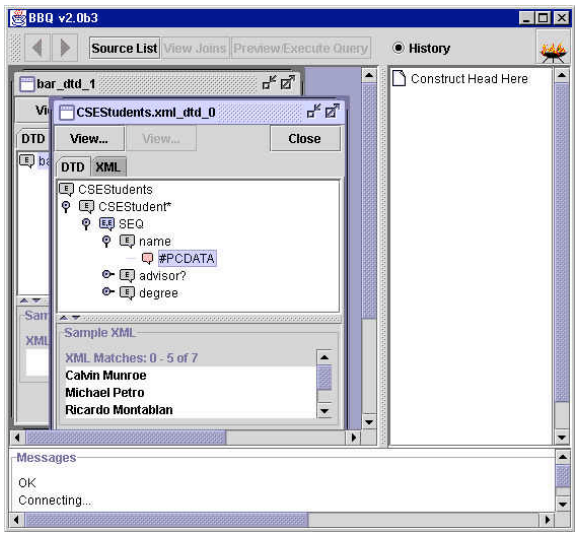
Here, one can observe the two main syntactic differences to XML-QL: (1) In XMAS, grouping is expressed by enclosing the variables on whose bindings the grouping is performed in curly braces and attaching them to the end of the subpattern that specifies the structure of the resulting instances. In the 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>
```

Fig. 4 Screenshot of BBQ's query editor
(Munroe and Papakonstantinou [215], © Kluwer, B.V.)




```
            </>
        </>
  </>
CONSTRUCT
  <results>
    [<result>
      $B
      [<book-author>
         $A
         <name> $N </name>
       </book-author>]
    </>]
  </>
```

No results on complexity or expressiveness of XMAS have been published.

BBQ [215] 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 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`

XML-RL. XML-RL [187] 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 $A \Leftarrow L_1, \ldots, L_n$ where A is used for construction and $L_1, \ldots, L_n$ 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

TQL. TQL [70, 93] is an XML query language based upon ambient logic [71], 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.

[70] 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 [70]) 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 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 [70], 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

Xcerpt. Xcerpt [28, 60, 61, 247, 248] is a query language designed after principles given in [57] 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 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 [77], 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" [62, 63]. 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 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:

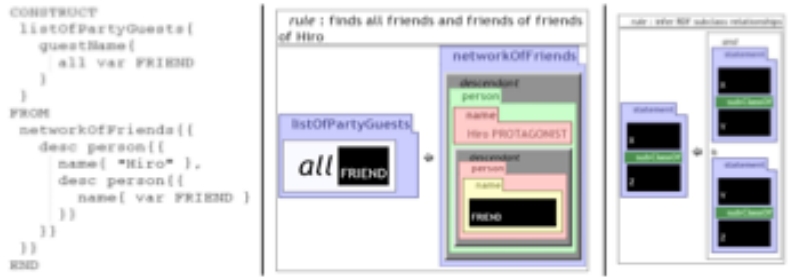
```
GOAL
results [
  all category [
    attributes [ id [ var ID ] ],
    average-number-of-authors [
      div( count( all var Author ), count( all var Book ) )
    ]
  ]
]
FROM
bookdata {{
  desc category {{ attributes {{ id [ var ID ] }} }},
  desc var Book → book {{
    attributes {{ type [ var ID ] }},
    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 ]
  ]
]
```

Fig. 5 Xcerpt and visXcerpt representation of a query




```
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 3.1x).

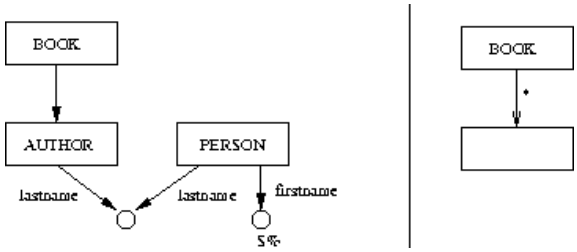
A visual language, called *visXcerpt* [29, 30], has been conceived as a visual rendering of textual Xcerpt programs, making it possible to freely switch during programming between the visual and textual view, or rendering, of a program (cf. Figure 5 showing a textual and visual representation of an Xcerpt query).

Static type checking methods have been developed for Xcerpt [55, 283] that are based on seeing tree 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 [63, 247]. A formal procedural semantics for Xcerpt has been proposed in [63] in the form of a a proof procedure. An implementation of this semantic in Haskell has been realized using Constraint Programming techniques [247]. The XQuery use case [76] has been worked out in Xcerpt (cf. [174] (in German) and [45]). Based on Xcerpt and extending it, a reactive language called XChange [58, 59] for updates and events on the Web is currently being developed.

Project page:
`http://www.xcerpt.org/`

Fig. 6 Graph representation of an XML-GL query
(Ceri et al. [75], © Elsevier, Inc.)



Implementation:
available from the project page
Online demonstration:
`http://demo.xcerpt.org` and, using visXcerpt, `http://visxcerpt.xcerpt.org/`

XML-GL. XML-GL [74, 75, 91] 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 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 [222] an extension of XML-GL in this direction is proposed.

Recently, a visual interface for XQuery, called XQBE [10, 47], based on XML-GL has been developed. Figure 7 shows the XQBE representation of the following XQuery expression (Query XMP-Q1 in [76]):

```
<bib>
{
  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 [47] (cf. Figure 8).

Fig. 7 XQBE Query
(Braga et al. [47], © Elsevier, Inc.)

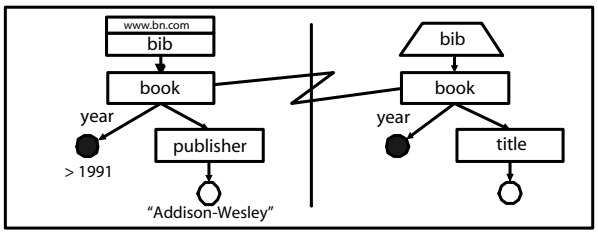


Fig. 8 Screenshot of XQBE's query editor
(Braga et al. [47], © Elsevier, Inc.)

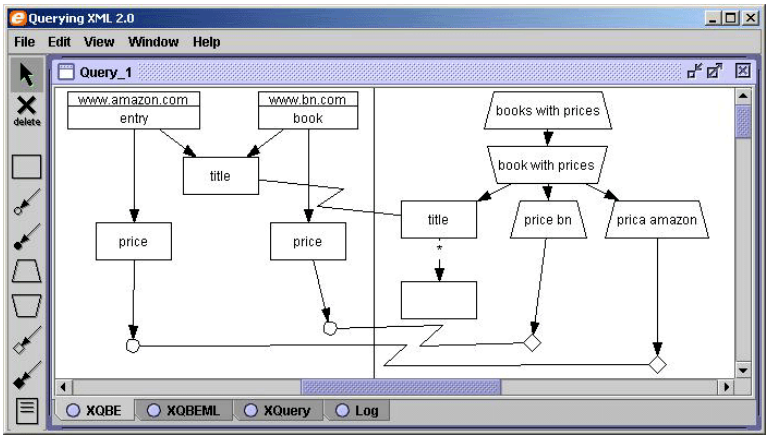
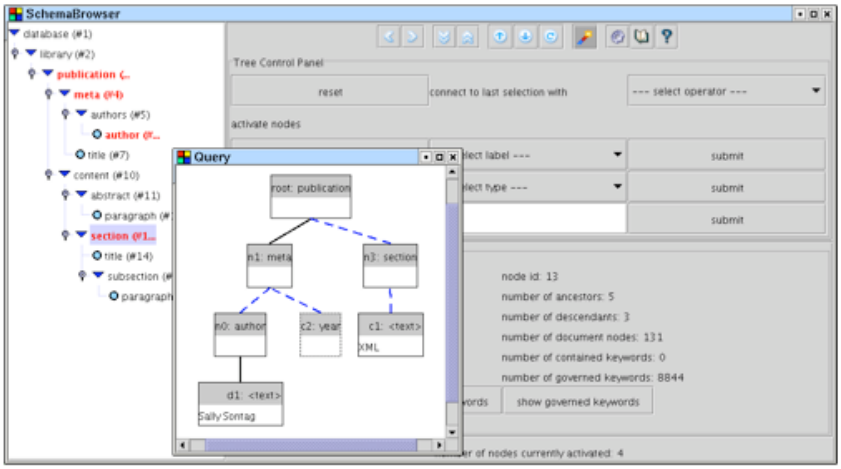


Fig. 9 Screenshot of X^2's query editor
(Meuss et al. [209], © Springer-Verlag)



Project page:
XQBE: `http://dbgroup.elet.polimi.it/xquery/XQBE.html`
Implementation:
XQBE: available from the project page
Online demonstration:
none

X^2's visual interface. X^2 [209] is a system for visual exploration and retrieval of XML databases. It provides an interactive environment for authoring visual queries, see Figure 9. The employed query language is rather restricted, but supports querying the order of elements and can be evaluated very efficiently (see [207]). 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" [207, 208] 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

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.

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".

Basic RDF Access: SquishQL and RDQL. The main objectives of SquishQL [212, 213] are ease-of-use and similarity to SQL. SquishQL relies on a query model for RDF influenced by [141]. 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."* [213]. 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 1 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 2 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 2 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 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 [212]

RDQL, a "RDF Data Query Language", is an evolution of the SquishQL versions SquishQL [213], and Inkling [212] influenced by rdfDB [140]. RDQL has been recently submitted to the W3C for standardisation [213, 251–253]. 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."* [251]. 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 [284]. RIDIQL supports updates and a transparent use of the inference abilities of the Jena Toolkit [136].

Project pages:
`http://www.hpl.hp.com/semweb/rdql.htm`
RDFStore: `http://rdfstore.sourceforge.net/`
Implementations:
Jena Toolkit [136, 251–253], RAP (RDF API for PHP) [221], PHP XML

Classes (`http://phpxmlclasses.sourceforge.net/`), RDFStore [240],
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/`) [146].

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 2.3, only Query 1 and (an approximation of) Query 2. Like SquishQL, RDQL does not support RDFS concepts, although at least one of its implementations, that given in the Jena Toolkit [136], 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.

SPARQL. SPARQL [239], a "Query Language for RDF" formerly called BrQL [238], has been developed by members of the W3C "RDF Data Access" Working Group. SPARQL is an extension of RDQL [251] designed according to requirements and use cases [87] 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. [267] 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 1 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 4 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 9 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

TriQL. TriQL extends RDQL by constructs supporting querying of named graphs [72], as introduced in TriG [40] 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

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.

RQL. RQL, the "RDF Query Language", is developed at ICS-FORTH [84, 158–161], 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`[11]. 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*:

[11] Assuming: `USING NAMESPACE books = &http://example.org/books-rdfs#`

```
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 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 8 is expressible in RQL as follows:
`SELECT X, Y  FROM Class{X}, subClassOf(X){Y}`.

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[12]) 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 $(X, \texttt{rdf:type}, \texttt{books:Writing})$ 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 $(X, \texttt{books:author}, Y)$ exists. RQL offers extended dot notation as used in OQL [73], for navigation in data and schema graphs. This is convenient for expressing Query 1:

```
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#
```

[12] i.e. deduceable by inference.

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 [191] 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 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 1 is already given in RQL above. Query 2 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 3, can often be concisely expressed in RQL, in particular if types are available:

```
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 4 is given above in the extension RVL of RQL. RQL is convenient for expressing aggregation queries, e.g., Query 5:

```
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 9, 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 9 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 [161]. 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 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/`

SeRQL. SeRQL [52, 67] 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 9: 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 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 3 cannot be expressed in SeRQL. Thanks to the `CONSTRUCT` clause, SeRQL, like RQL, can express restructuring and simple inference queries, e.g., 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 cannot be expressed in SeQL (according to [67], 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[13]: `http://gollem.swi.psy.uva.nl/twiki/pl/bin/view/Library/SeRQL`
Online demonstrations:
`http://www.openrdf.org/demo.jsp`

eRQL. eRQL [271] 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*".[14] 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 2):

[13] Using the Semantic Web library of SWI Prolog `http://www.swi-prolog.org/`.
[14] `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 2.3 cannot be expressed in eRQL. The extraction query Query 2 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" [267]. 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

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. [245, 266, 277]) can be implemented with a few additional functions and/or by normalising the data before querying.

XQuery for RDF: The "Syntactic Web Approach". [242, 245] propose to rely on the XML Query Language *XQuery* (cf. Section 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 1 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 [231]. [257] suggests a similar approach.

Project page:
none
Implementation(s):
not publicly available
Online demonstration:
none

XSLT for RDF: TreeHugger and RDF Twig. Similar in spirit to the Syntactic Web Approach [242, 245], TreeHugger [266] 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" [50], 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 1 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 [277] 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 4.1.

Query 1 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

Versa. Developed as part of the Python-based *4Suite* XML and RDF toolkit[15], Versa [219, 220, 223] 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 1 can be expressed as follows:

```
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')"
```

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

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 1 demonstrates. In contrast to most RDF query languages, Versa allows the extraction of RDF subgraphs of arbitrary sizes, as required by Query 2. Reduction queries can be expressed in Versa, e.g., using negation or set difference. Query 3 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 4 and 9 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 5 can be expressed as follows in Versa:

```
max(filter(all(),
      ". - books:author -> books:authorName -> eq('Julius Caesar')"
      )
    - books:year -> *)
```

Query 6 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

Path-Based Access to RDF: RDF Path, RPath, RxPath, RxSLT, and RxUpdate. [229] 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 1 is not expressible, since related information cannot be selected. A variation of Query 2, *"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 [201] 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 2 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 [50]). 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[16], 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"* [263]. 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" [262],
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." [262], 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." [262].

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 [50]). 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 2 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
```

[16] P`http://rx4rdf.liminalzone.org/rx4rdf`

Based on RxPath, two languages have been defined, RxSLT [264] and RxUpdate [265]. RxSLT is *"syntactically identical to XSLT 1.0"* [264], but uses RxPath instead of XPath 1.0. RxUpdate is syntactically very similar to XUpdate [185], 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.

Project page:
`http://rx4rdf.liminalzone.org/rx4rdf`
Implementation(s):
Cf. project page (prototype in Python)
Online demonstration:
none

RDFT and the Query Language of Nexus: XSLT-Style RDF Query Languages. RDFT [95] 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" [50]. 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 1 (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 [95] 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

[1] 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. [1] 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`. [1] 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 1 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 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 [162], an XQuery-style RDF Query Language, is inspired from XQuery 1.0 [41], aiming at simplicity and flexibility. XsRQL departs from XQuery as follows: (1) The data model is adapted from RDF ([162] 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 `@`.[17]

Query 1 can be approximated in XsRQL as follows:

```
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

[17] In XPath, `@` indicate (flat) XML attributes. Since RDF properties are structured, in XsRQL a path expression may follow a `@` step.

4.4 Metalog: Querying in Controlled English

Metalog [195–197] 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 1 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

4.5 Query Languages with Reactive Rules.

Algae. Algae[18] 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

[18] Also called "Algae2". This survey follows [237] and retains the name "Algae".

?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’’.

Table 1. Answer to Query 1

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 [133]. 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 1 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 1.

Query 2 and Query 4 cannot be expressed in Algae due to the lack of closure, recursion, and negation. Queries 5 and 6 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' [236].

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/`
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`

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 1 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

WQL. WQL, Wilbur Query Language, is the name given in [182] to query primitives of Ivanhoe [181], a frame-based API inspired from [153, 178] for the Nokia Wilbur Toolkit [183], a collection of APIs for XML, RDF, and DAML written in CLOS, Common Lisp Object System [179].

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 [144]:

- 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.[19]
- WQL+CL+inference, an extension of WQL+CL, with a data store providing inferencing based upon the "transparent" (or "hidden") inference extensions described in [180].

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

4.6 Deductive Query Languages

N3QL. N3QL, sketched in [34], is derived from the rule fragment of Notation 3 [35] (shorthand N3), a syntax for and extension of RDF with variables, rules, and quoting for easy expression of statements about statements. N3QL differs

[19] It is unclear whether WQL+CL restricts Common Lisp.

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 1 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 }.
```

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.

[34] 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. [34] 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

R-DEVICE. R-DEVICE [20] is a *"deductive object-oriented knowledge-base system for querying and reasoning about RDF metadata."*[20] It is a reimplementation of the X-DEVICE language [19] 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 1 can be expressed as follows:

```
(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 2 and Query 8. Query 2 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

[20] `http://lpis.csd.auth.gr/systems/r-device.html`

TRIPLE TRIPLE [147, 259, 260] is a rule-based query, inference, and transformation language for RDF. TRIPLE is based upon ideas published in [96]. TRIPLE's syntax is close to F-Logic [170]. F-Logic is convenient for querying semi-structured data, e.g., XML and RDF, as it facilitates describing schemaless or irregular data [188]. Other approaches to querying XML and/or RDF are XPathLog and the ontology management platform Ontobroker[21]. 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) [170]. 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 [176, 177], the authors of TRIPLE claim in [260] that TRIPLE is well-suited to query non-RDF meta-data. This can be questioned, especially if, in spite of [126], one considers the rather awkward mappings of Topic Maps into RDF proposed so far.

TRIPLE differs from Horn logic and Logic Programming as follows [260]:

- 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 [119] can be used for traversing several properties.
- 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 1 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 1 is properly expressed below.

TRIPLE's rules give rise to specify properties of RDF. [260] gives the following implementation of a part of RDFS's semantics:

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

```
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 1 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 2.3 can be expressed in TRIPLE. Aggregation queries cannot be expressed in TRIPLE, for the language does not support aggregation.

[260] 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/`[22]

Xcerpt. Xcerpt [28, 60, 61, 248], 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 3.2. 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 2.3 can be expressed in Xcerpt's both on the XML serialization (cf. Section 3.2) and on the RDF serialization of the sample data from Section 2.2. The following Xcerpt programs show solutions for the queries against the RDF serialization.

[44] 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 [
```

[22] Not functioning at the time of writing.

```
      var Author:uri{}, "books:authorName":uri{}, var AuthorName
    ]
  }
END
```

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 [148] (cf. [44] 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 [242] 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 [25], the official XML serialisation, are very complex and lengthy due to the high degree of flexibility RDF/XML allows. They can be found in [44], similar functions for parsing RDF/XML in XQuery are described in [245]. The following rules parse RDF data serialised in the RXR (Regular XML RDF) format [11], 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 [44].

```
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 [44].

4.7 Other RDF Query Languages

RDF-QBE [241] is inspired from QBE [289, 290], the database query language that introduced the celebrated "Query by Example" paradigm. An RDF graph, expressed in the syntax of Notation 3 [35]), is used to describe query patterns, variables are expressed as blank nodes that, according to [172] 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.[23] 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 2.3 can be expressed.

Project page:
none
Implementation:
described in [241] but not publicly available
Online demonstration:
none

[23] [241] (wrongly) suggests that this restriction reduces query-answering to tree matching because the data queried is not necessarily tree-shaped.

RDFQL. RDFQL is the query language of RDF Gateway [154], 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 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[24] "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 1 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}
```

[24] I.e., identified, e.g., by an URI.

```
  and {[books:authorName] ?author ?authorName}
ORDER BY ?authorName DESC;
```

Project page:
`http://www.intellidimension.com/`
Implementations:
RDF Gateway
Cf. project page for a limited, non-commercial use
Online demonstration:
none[25]

5 Topic Maps Query Languages

5.1 tolog: Logic Programming for Topic Maps

tolog [121–124] is the query language of the Ontopia Knowledge Suite[26]. 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"`.[27] 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 [123]) 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 [122, 124] 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 2 can only be implemented only using these predicates:

```
select $RELATED from
  title($BOOK, "Bellum Civile"),
```

[25] However, the project page implemented in RDF Gateway is a show case.
[26] `http://www.ontopia.net/solutions/products.html`
[27] The prefix `i` serves to distinguish different identifiers.

```
  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 [123]:

```
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 [122]. tolog has constructs for aggregation and sorting (although deemed insufficient [122]), paged queries using `limit` and `offset` as in SQL, and a module concept. Thanks to tolog's (possibly recursive) rules, Queries 7 and 8 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`[28]
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`[29]

[28] Tutorial.
[29] The demonstrator does not seem to support testing tolog queries.

5.2 AsTMA?: Functional Style Querying of Topic Maps

AsTMa? [14, 16] is a functional query language in the style of XQuery [41]. AsTMa? offers several path languages for accessing data in topic maps. With AsTMa?, answers can be re-structured, yielding new XML documents.

Query 1 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 1 can also be implemented as follows, using path expressions for accessing topics and associations:

```
<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/`

5.3 Toma: Querying Topic Maps inspired from SQL

Toma [175, 234] 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 1 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 3 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

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. [15] for an overview of a plea for the inclusion of path navigation in the upcoming ISO Topic Maps query language.

XTMPath [17] is a path-based query language relying on the XTM [232] 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[30], viz., the DTD of XTM DTD [17], 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

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 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 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:

[30] Document Type Definition, cf. [48].

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 [76], 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" [2] 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.[31]

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.

Acknowledgements

The authors are thankful to Renzo Orsini, Ian Horrocks, Michael Kraus, and Oliver Bolzer for stimulating discussions and useful suggestions during the production of the report [56], that has been an important input for this overview.

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`).

[31] Indeed, materialising conclusions from all possible combinations of Web sites is infeasible.

Bibliography

[1] Langdale Consultants . Nexus Query Language. Online only, 2000.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[3] 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.

[4] J. Alferes, W. May, and P. Patranjan. *State of the Art on Evolution and Reactivity*, 2004.

[5] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, D. McBeath, M. Rys, and J. Shanmugasundaram. *XQuery and XPath Full-Text*. W3C, 2004. URL `http://www.w3.org/TR/xquery-full-text-requirements/`.

[6] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In *Proc. Int. World Wide Web Conf.*, 2004.

[7] 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.

[8] 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.

[9] 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.

[10] 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. ISBN 1-58113-624-2.

[11] D. Backett. Modernising Semantic Web Markup. In *Proc. XML Europe*, April 2004.

[12] 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.

[13] J. Bailey. Transformation and Reaction Rules for Data on the Web. In *Proc. Australasian Database Conference*, 2005.

[14] R. Barta. AsTMa? Tutorial. Technical report, Bond University, 2003.

[15] R. Barta. Path Language for Topic Maps: Full speed ahead? Online only, 2004.

[16] R. Barta. AsTMa= Language Definition. Online only, 2007.

[17] R. Barta and J. Gylta. *XTM::Path*, 2002.

[18] 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.

[19] N. Bassiliades and I. Vlahavas. Intelligent Querying of Web Documents Using a Deductive XML Repository. In *Proc. Hellenic Conference on Artificial Intelligence*, April 2002.
[20] 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.
[21] 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.
[22] 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. ISBN 1-58113-844-X.
[23] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. *OWL Web Ontology Language—Reference*. W3C, 2004. URL `http://www.w3.org/TR/owl-ref/`.
[24] D. Beckett. *Turtle - Terse RDF Triple Language*, February 2004.
[25] D. Beckett and B. McBride. *RDF/XML Syntax Specification (Revised)*. W3C, 2004. URL `http://www.w3.org/TR/rdf-syntax-grammar/`.
[26] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *Proc. International Conference on Database Theory*, 2003.
[27] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proc. International Conference on Functional Programming*, 2003.
[28] 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.
[29] 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.
[30] 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.
[31] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. *XML Path Language (XPath) 2.0*. W3C, 2005.
[32] 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.
[33] A. Berlea and H. Seidl. Binary Queries for Document Trees. *Nordic Journal of Computing*, 11(1):41–71, 2004.
[34] T. Berners-Lee. N3QL—RDF Data Query Language. Online only, 2004.
[35] T. Berners-Lee. Notation 3, an RDF language for the Semantic Web. Online only, 2004.
[36] T. Berners-Lee. Semantic Web Road Map. Online only, 2004.
[37] 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.

[38] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. *Information Systems*, 27(1):21–39, 2002. ISSN 0306-4379.

[39] P. Biron and A. Malhotra. *XML Schema Part 2: Datatypes*. W3C, 2001. URL `http://www.w3.org/TR/xmlschema-2/`.

[40] C. Bizer. The TriG Syntax. Online only, April 2004.

[41] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. *XQuery 1.0: An XML Query Language*. W3C, 2005.

[42] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2 2005.

[43] H. Boley, B. Grosof, M. Sintek, S. Tabet, and G. Wagner. RuleML Design. Online only, 2002.

[44] O. Bolzer. Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/Master thesis, University of Munich, 2 2005.

[45] 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.

[46] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. Int. Conf. on Data Engineering*, page 403. IEEE Computer Society, 2002.

[47] D. Braga, A. Campi, S. Ceri, and E. Augurusa. XQuery by Example. In *Proc. Int. World Wide Web Conf.*, 2003.

[48] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C, 2004. URL `http://www.w3.org/TR/REC-xml/`.

[49] J.-M. Bremer and M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In *Int. Workshop on the Web and Databases*, 2002.

[50] D. Brickley. RDF: Understanding the Striped RDF/XML Syntax. Online only, October 2001.

[51] D. Brickley, R. Guha, and B. McBride. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C, 2004. URL `http://www.w3.org/TR/rdf-schema/`.

[52] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003.

[53] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[54] 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. ISBN 1-58113-724-9.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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. ISBN 0-89791-794-4.

[65] 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. ISBN 3-540-76086-5.

[66] 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.

[67] A. b.v. and S. A. Ltd. *The SeRQL query language*, chapter 5. Aduna b.v., Sirma AI Ltd., 2002.

[68] 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.

[69] 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.

[70] 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. ISSN 0960-1295.

[71] 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. ISBN 1-58113-125-9.

[72] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named Graphs, Provenance and Trust. Technical Report HPL-2004-57, HP Labs, 2004.
[73] 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.
[74] 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.
[75] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents. In *Proc. Int. World Wide Web Conf.*, 1999.
[76] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Cases*. W3C, 2005.
[77] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. *XML Query (XQuery) Requirements*. W3C, 2003.
[78] D. Chamberlin and J. Robie. XQuery Update Facility Requirements. Working draft, W3C, 2005.
[79] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. Workshop on Web and Databases*, 2000.
[80] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *Proc. Workshop on the Web and Databases*, 2002.
[81] 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.
[82] 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.
[83] 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.
[84] 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.
[85] J. Clark. *XSL Transformations (XSLT) Version 1.0*. W3C, 1999.
[86] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0*. W3C, 1999.
[87] K. Clark. *RDF Data Access Use Cases and Requirements*. W3C, 2004.
[88] 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.
[89] 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. ISSN 1532-2882.

[90] 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.

[91] S. Comai, E. Damiani, and P. Fraternali. Computing Graphical Queries over XML Data. *ACM Transactions on Information Systems*, 19(4):371–430, 2001. ISSN 1046-8188.

[92] S. Comai, S. Marrara, and L. Tanca. XML Document Summarization: Using XQuery for Synopsis Creation. In *Proc. Int. Workshop on Database and Expert Systems Applications*, 2004.

[93] 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.

[94] J. Cowan and R. Tobin. *XML Information Set (Second Edition)*. W3C, 2004. URL `http://www.w3.org/TR/2004/REC-xml-infoset-20040204/`.

[95] I. Davis. RDF Template Language 1.0. Online only, September 2003.

[96] 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.

[97] D. DeHaan, D. Toman, M. P. Consens, and M. T. zsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. ACM SIGMOD Conf.*, pages 623–634. ACM Press, 2003. ISBN 1-58113-634-X.

[98] 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.

[99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proc. Int. World Wide Web Conf.*, 1999.

[100] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[101] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proc. Int. Workshop on Knowledge Representation meets Databases*, 2001.

[102] C. Dong and J. Bailey. Optimization of XML Transformations Using Template Specialization. In *Proc. Int. Conf. on Web Information Systems Engineering*, 2004.

[103] C. Dong and J. Bailey. Static Analysis of XSLT Programs. In *Proc. Australasian Database Conf.*, pages 151–160. Australian Computer Society, Inc., 2004. ISBN 1-111-11111-1.

[104] D. Draper, P. Frankhauser, M. Fernndez, A. Malhotra, K. Rose, M. Rys, J. Simon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Working draft, W3C, 2 2005.

[105] D. Eastlake and A. Panitz. Reserved Top Level DNS Names. RFC 2606, IETF, 1999.

[106] A. Eisenberg and J. Melton. An early Look at XQuery. *SIGMOD Record*, 31(4):113–120, 2002. ISSN 0163-5808.

[107] A. Eisenberg and J. Melton. An early Look at XQuery API for Java^{TM}(XQJ). *SIGMOD Record*, 33(2):105–111, 2004. ISSN 0163-5808.
[108] D. Fallside. *XML Schema Part 0: Primer*. W3C, 2001. URL `http://www.w3.org/TR/xmlschema-0/`.
[109] P. Fankhauser. XQuery Formal Semantics: State and Challenges. *SIGMOD Record*, 30(3):14–19, 2001. ISSN 0163-5808.
[110] P. Fankhauser and P. Lehti. XQuery by the book: The IPSI XQuery Demonstrator. In *XML Conference & Exhibition*, 2002.
[111] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C, 2004.
[112] M. Fernndez, J. Simon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0 : The Galax Experience. In *Proc. Int. Conf. on Very Large Databases*, 2003.
[113] 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.
[114] 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.
[115] D. Florescu, A. Grnhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proc. International World Wide Web Conference*, May 2002.
[116] D. Florescu, A. Grnhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. *Computer Networks*, 42(5), 2003.
[117] 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. ISSN 1066-8888.
[118] 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.
[119] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In *Proc. International Conference on Very Large Databases*, 1994.
[120] 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.
[121] L. Garshol. tolog—A topic map query language. In *Proc. XML Europe*, 2001.
[122] L. Garshol. Extending tolog—Proposal for tolog 1.0. In *Proc. Extreme Markup Languages*, 2003.
[123] L. Garshol. tolog 0.1. Technical report, Ontopia, 2003.
[124] L. Garshol. tolog–Language tutorial. Online only, 2004.
[125] L. Garshol. The Linear Topic Map Notation. Online only, 2007.
[126] L. M. Garshol. Living with Topic Maps and RDF. Online only, 2003.
[127] 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.

[128] 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. ISBN 0-7695-1483-9.

[129] 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.

[130] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. International Conference on Very Large Databases*, 2002.

[131] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proc. ACM Symposium on Principles of Database Systems*, 2003.

[132] G. Gottlob, C. Koch, and R. Pichler. XPath Query Evaluation: Improving Time and Space Efficiency. In *Proc. International Conference on Data Engineering*, 2003.

[133] J. Grant and D. Backett. *RDF Test Cases*. W3C, February 2004.

[134] S. Groppe and S. Bttcher. XPath Query Transformation based on XSLT Stylesheets. In *Proc. Int. Workshop on Web Information and Data Management*, pages 106–110. ACM Press, 2003. ISBN 1-58113-725-7.

[135] P. Grosso, E. Maier, J. Marsh, and N. Walsh. *XPointer Framework*. W3C, 2003. URL `http://www.w3.org/TR/xptr-framework/`.

[136] H. L. S. W. R. Group. Jena – A Semantic Web Framework for Java. Online only, 2004.

[137] T. Grust. Accelerating XPath Location Steps. In *Proc. ACM SIGMOD Conf.*, 2002.

[138] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004. ISSN 0362-5915.

[139] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. Int. Conf. on Very Large Databases*, 2004.

[140] R. Guha. rdfDB Query Language. Online only, 2000.

[141] R. Guha, O. Lassila, E. Miller, and D. Brickley. Enabling Inferencing. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[142] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proc. ACM SIGMOD Conf.*, 2003.

[143] Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evaluation. In *Proc. Asia Pacific Web Conference*, 2004.

[144] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *Proc. International Semantic Web Conference*, 2004.

[145] 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.

[146] S. Harris and N. Gibbins. 3store: Efficient Bulk RDF Storage. In *Proc. International Workshop on Practical and Scalable Semantic Systems*, 2003.

[147] A. Harth. Triple Tutorial. Online only, 2004.

[148] P. Hayes and B. McBride. *RDF Semantics*. W3C, 2004. URL `http://www.w3.org/TR/rdf-mt/`.

[149] J. Hidders. Satisfiability of XPath Expressions. In *Int. Workshop on Databse Programming Languages*, 2003.

[150] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. *SWRL: A Semantic Web Rule Language—Combining OWL and RuleML*. W3C, 2004. URL `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`.

[151] I. Horrocks, F. van Harmelen, and P. Patel-Schneider. *DAML+OIL*. Joint US/EU ad hoc Agent Markup Language Committee, 2001. URL `http://www.daml.org/2001/03/daml+oil-index.html`.

[152] H. Hosoya and B. Pierce. XDuce: A Typed XML Processing Language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[153] J. Hynynen and O. Lassila. On the Use of Object-Oriented Paradigm in a Distributed Problem Solver. *AI Communications*, 2(3):142–151, 1989.

[154] Intellidimension. RDF Gateway. Online only, 2004.

[155] *ISO/IEC 13250 Topic Maps*. International Organization for Standardization, 1999. URL `http://www.y12.doe.gov/sgml/sc34/document/0322_files/iso13250-2nd-ed-v2.pdf`.

[156] 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. ISBN 1-58113-449-5.

[157] 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.

[158] 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.

[159] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF Descriptions for Community Web Portals. In *Proc. Journees Bases de Donnees Avancees*, 2001.

[160] 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.

[161] 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. ISBN 3-540-00375-4.

[162] H. Katz. XsRQL: an XQuery-style Query Language for RDF. Online only, 2004.

[163] 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.

[164] M. Kay. *XPath2.0 Programmer's Reference*. John Wiley, 8 2004.
[165] M. Kay. *XSLT 2.0 Programmer's Reference*. John Wiley, 3rd edition, 8 2004.
[166] M. Kay. XSLT and XPath Optimization. In *XML Europe*, 2004.
[167] M. Kay. *XSL Transformations (XSLT) Version 2.0*. W3C, 2005.
[168] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 Serialization. Working draft, W3C, 2 2005.
[169] S. Kepser. A Simple Proof of the Turing-Completeness of XSLT and XQuery. In *Proc. Extreme Markup Languages*, 2004.
[170] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object Oriented and Frame Based Languages. *Journal of ACM*, 42:741–843, 1995.
[171] C. Kirchner, Z. Oian, P. Singh, and J. Stuber. Xemantics: a Rewriting Calculus-Based Semantics of XSLT. Technical Report A01-R-386, LORIA, 2002.
[172] G. Klyne, J. Carroll, and B. McBride. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C, 2004. URL `http://www.w3.org/TR/rdf-concepts/`.
[173] 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.
[174] S. Kraus. Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web. Diplomarbeit/Master thesis, University of Munich, 2004.
[175] R. Ksiezyk. Answer is just a question [of matching Topic Maps]. In *Proc. XML Europe*, 2000.
[176] M. Lacher and S. Decker. On the Integration of Topic Maps and RDF Data. In *Proc. Extreme Markup Languages*, 2001.
[177] M. Lacher and S. Decker. RDF, Topic Maps, and the Semantic Web. *Markup Languages: Theory and Practice*, 3(3):313–331, December 2001.
[178] 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.
[179] O. Lassila. Enabling Semantic Web Programming by Integrating RDF and Common Lisp. In *Proc. Semantic Web Working Symposium*, july 2001.
[180] O. Lassila. Taking the RDF Model Theory Out for a Spin. In *Proc. Semantic Web Working Symposium*, June 2002.
[181] O. Lassila. Ivanhoe: an RDF-Based Frame System. Online only, 2004.
[182] O. Lassila. Wilbur Query Language Comparison. Online only, 2004.
[183] O. Lassila. Wilbur Semantic Web Toolkit. Online only, 2004.
[184] O. Lassila and R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C, 1999. URL `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.
[185] A. Laux and L. Martin. *XUpdate—XML Update Language*. XML:DB Initiative, 2000. URL `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`.

[186] J. Liu and M. Vincent. Query translation from XSLT to SQL. In *Proc. Int. Database Engineering and Applications Symposium*, 2003.

[187] M. Liu. A Logical Foundation for XML. In *Proc. International Conference on Advanced Information Systems Engineering*. Springer-Verlag, 2002.

[188] 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.

[189] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. *A Brief Introduction to XMAs*. Database Group, University of California, San Diego, 1999.

[190] 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.

[191] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proc. International Semantic Web Conference*, October 2003.

[192] D. Maier. Database Desiderata for an XML Query Language. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[193] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2 2005.

[194] F. Manola, E. Miller, and B. McBride. *RDF Primer*. W3C, 2004. URL `http://www.w3.org/TR/rdf-primer/`.

[195] M. Marchiori, A. Epifani, and S. Trevisan. Metalog v2.0: Quick User Guide. Technical report, W3C, 2004.

[196] M. Marchiori and J. Saarela. Query + Metadata + Logic = Metalog. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[197] M. Marchiori and J. Saarela. Towards the Semantic Web: Metalog. Online only, 1999.

[198] 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.

[199] 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.

[200] M. Marx. XPath with Conditional Axis Relations. In *Proc. Extending Database Technology*, 2004.

[201] 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.

[202] N. May, S. Helmer, and G. Moerkotte. Quantifiers in XQuery. In *Proc. Int. Conf. on Web Information Systems Engineering*, 2003.

[203] 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.

[204] D. McGuinness and F. van Harmelen. *OWL Web Ontology Language—Overview*. W3C, 2004. URL `http://www.w3.org/TR/owl-features/`.

[205] E. Meijer, W. Schulte, and G. Bierman. Programming with Circles, Triangles and Rectangles. In *Proc. XML Conference and Exhibition*, 2003.

[206] E. Meijer and M. Shields. XMLambda: A functional language for constructing and manipulating XML documents. Online only, 1999.

[207] 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. ISSN 1046-8188.

[208] 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. ISBN 3-540-41456-8.

[209] 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.

[210] H. Meyer, I. Bruder, A. Heuer, and G. Weber. The Xircus Search Engine. In *INEX Workshop*, pages 119–124, 2002.

[211] 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. ISBN 1-58113-507-6.

[212] L. Miller. Inkling: RDF query using SquishQL. Online only, 2004.

[213] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *Proc. International Semantic Web Conference*, June 2002.

[214] 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. ISBN 1-58113-214-X.

[215] 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. ISBN 0-7923-7835-0.

[216] 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. ISBN 1-58113-738-9.

[217] M. Nilsson, W. Siberski, and J. Tane. Edutella Retrieval Service: Concepts and RDF Syntax. Online only, June 2004.

[218] M. Odersky. Report on the Programming Language Scala. Technical report, Ecole Polytechnique Federale de Lausanne, 2002.

[219] U. Ogbuji. Thinking XML: Basic XML and RDF techniques for knowledge management: Part 6: RDF Query using Versa. Online only, April 2002.

[220] U. Ogbuji. Versa by example. Online only, 2004.

[221] R. Oldakowski and C. Bizer. RAP: RDF API for PHP. In *Proc. International Workshop on Interpreted Languages*, 2004.

[222] 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. ISSN 0306-4379.

[223] M. Olson and U. Ogbuji. Versa Specification. Online only, 2003.

[224] D. Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/Ph.D. thesis, University of Munich, 1 2005.

[225] 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.

[226] 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. ISBN 1-58113-859-8.

[227] 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.

[228] N. Onose and J. Simeon. XQuery at your Web Service. In *Proc. Int. World Wide Web Conf.*, pages 603–611. ACM Press, 2004. ISBN 1-58113-844-X.

[229] S. Palmer. Pondering RDF Path. Online only, 2003.

[230] 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.

[231] P. Patel-Schneider and J. Simeon. The Yin/Yang Web: XML Syntax and RDF Semantics. In *Proc. International World Wide Web Conference*, May 2002.

[232] S. Pepper and G. Moore. *XML Topic Maps (XTM) 1.0*. TopicMaps.org, 2001. URL `http://www.topicmaps.org/xtm/index.html`.

[233] 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. ISBN 1-58113-432-0.

[234] R. Pinchuk. Toma - Topic Map Query Language. Online only, 2004.

[235] M. Plusch. *Water: Simplified Web Services and XML Programming*. Wiley, 2002. ISBN 0764525360.

[236] E. Prud'hommeaux. Algae Extension for Rules. Online only, 2004.

[237] E. Prud'hommeaux. Algae RDF Query Language. Online only, 2004.

[238] E. Prud'hommeaux and A. Seaborne. BRQL – A Query Language for RDF. Online only, 2004.

[239] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, February 2005.

[240] A. Reggiori and D.-W. van Gulik. RDFStore—Perl API for RDF Storage. Online only, 2004.

[241] D. Reynolds. RDF-QBE: a Semantic Web Building Block. Technical Report HPL-2002-327, HP Labs, 2002.

[242] J. Robie. The Syntactic Web: Syntax and Semantics on the Web. In *Proc. XML Conference and Exposition*, December 2001.

[243] J. Robie. Updates in XQuery. In *XML Conference & Exhibiton*, 2001.

[244] J. Robie, E. Derksen, P. Frankhauser, E. Howland, G. Huck, I. Macherius, M. Murata, M. Resnick, and H. Schning. XQL (XML Query Language). Online only, 1999.

[245] 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.

[246] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[247] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[248] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, August 2004.

[249] S. Schott and M. L. Noga. Lazy XSL Transformations. In *Proc. ACM Symposium on Document Engineering*, pages 9–18. ACM Press, 2003. ISBN 1-58113-724-9.

[250] T. Schwentick. XPath Query Containment. *SIGMOD Record*, 2004.

[251] A. Seaborne. RDQL – A Query Language for RDF. Online only, January 2004.

[252] A. Seaborne. RDQL – RDF Data Query Language. Online only, 2004.

[253] A. Seaborne. A Programmer's Introduction to RDQL. Online only, 2002 April.

[254] D. Seipel. Processing XML-Documents in Prolog. In *Workshop on Logic Programming*, 2002.

[255] D. Seipel and J. Baumeister. Declarative Methods for the Evaluation of Ontologies. *KI–Knstliche Intelligenz*, 4:51–57, 2004.

[256] 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.

[257] R. Shearer. REX evaluation. Online only, 2004.

[258] J. E. Simpson. *XPath and XPointer*. O'Reilly, 1st edition, 9 2002.

[259] M. Sintek and S. Decker. TRIPLE—An RDF Query, Inference, and Transformation Language. In *Proc. Deductive Database and Knowledge Management*, October 2001.

[260] M. Sintek and S. Decker. TRIPLE—A Query, Inference, and Transformation Language for the Semantic Web. In *Proc. International Semantic Web Conference*, June 2002.

[261] M. Smith, C. Welty, and D. McGuinness. *OWL Web Ontology Language—Guide*. W3C, 2004. URL `http://www.w3.org/TR/owl-guide/`.

[262] A. Souzis. RxPath. Online only, 2004.

[263] A. Souzis. RxPath Specification Proposal. Online only, 2004.

[264] A. Souzis. RxSLT. Online only, 2004.

[265] A. Souzis. RxUpdate. Online only, 2004.

[266] D. Steer. TreeHugger 1.0 Introduction. Online only, 2003.

[267] P. Stickler. CBD—Concise Bounded Description. Online only, 2004.

[268] I. Tatarinov and A. Halevy. Efficient Query Reformulation in peer Data Management Systems. In *Proc. ACM SIGMOD Conf.*, pages 539–550. ACM Press, 2004. ISBN 1-58113-859-8.

[269] J. Tennison. *XSLT and XPath On The Edge*. John Wiley, 10 2001.

[270] 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. ISBN 1-58113-497-5.

[271] K. Tolle and F. Wleklinski. easy RDF Query Language (eRQL). Online only, 2004. URL `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/eRQL/`.

[272] A. Tozawa. Towards Static Type Checking for XSLT. In *Proc. ACM Symposium on Document Engineering*, pages 18–27. ACM Press, 2001. ISBN 1-58113-432-0.

[273] A. Trombetta and D. Montesi. Equivalences and Optimizations in an Expressive XSLT Fragment. In *Proc. Int. Database Engineering and Applications Symposium*, 2004.

[274] L. Villard and N. Layada. An Incremental XSLT Transformation Processor for XML Document Manipulation. In *Proc. Int. World Wide Web Conf.*, pages 474–485. ACM Press, 2002. ISBN 1-58113-449-5.

[275] P. Wadler. Two semantics for XPath. Online only, 2000.

[276] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation. In *Proc. International Conference on Functional Programming*, 1999.

[277] N. Walsh. RDF Twig: accessing RDF graphs in XSLT. In *Proc. Extreme Markup Languages*, 2003.

[278] 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.

[279] S. Waworuntu and J. Bailey. XSLTGen: A System for Automatically Generating XML Transformations via Semantic Mappings. In *Proc. Int. Conf. on Conceptual Modeling*, 2004.

[280] 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.

[281] N. Wiegand. Investigating XQuery for Querying across Database Object Types. *SIGMOD Record*, 31(2):28–33, 2002. ISSN 0163-5808.

[282] U. Wiger. XMErl—Interfacing XML and Erlang. In *Proc. International Erlang User Conference*, 2000.

[283] 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.

[284] C. Wilper. RIDIQL Reference. Online only, 2004.

[285] P. T. Wood. On the Equivalence of XML Patterns. In *Proc. Int. Conf. on Computational Logic*, pages 1152–1166. Springer-Verlag, 2000. ISBN 3-540-67797-6.

[286] C. Zaniolo. The Database Language GEM. In *Proc. ACM SIGMOD Conf.*, 1983.

[287] 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. ISBN 1-58113-634-X.

[288] 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. ISBN 1-58113-593-9.

[289] M. Zoof. Query By Example. In *Proc. AFIPS National Computer Conference*, 1975.

[290] M. Zoof. Query By Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.