RDF Querying: Language Constructs and Evaluation Methods Compared

Tim Furche[1], Benedikt Linse[1], François Bry[1], Dimitris Plexousakis[2,3], and Georg Gottlob[4]

[1] Institute for Informatics,University of Munich,
Oettingenstraße 67, 80538 München, Germany
http://www.pms.ifi.lmu.de/

[2] Department of Computer Science, University of Crete
Vassilika Vouton, P.O.Box 1385, GR 711 10 Heraklion, Crete, Greece
http://www.ics.forth.gr/isl/people/people_individual.jsp?Person_ID=5

[3] Information Systems Laboratory, Institute of Computer Science, FORTH
Vassilika Vouton, P.O.Box 1385, GR 711 10 Heraklion, Crete, Greece

[4] Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
http://web.comlab.ox.ac.uk/oucl/people/georg.gottlob.html

Abstract. This article is firstly an introduction into query languages for the Semantic Web, secondly an in-depth comparison of the languages introduced. Only RDF query languages are considered because, as of the writing of this paper, query languages for other Semantic Web data modeling formalisms, especially OWL, are still an open research issue, and only a very small number of, furthermore incomplete, proposals for querying Semantic Web data modeled after other formalisms than RDF exist. The limitation to a few RDF query languages is motivated both by the objective of an in-depth comparison of the languages addressed and by space limitations. During the three years before the writing of this article, more than three dozen proposals for RDF query languages have been published! Not only such a large number, but also the often immature nature of the proposals makes the focus on few, but representative languages a necessary condition for a non-trivial comparison.

For this article, the following RDF query languages have been, admittedly subjectively, selected: Firstly, the “relational” or “pattern-based” query languages SPARQL, RQL, TRIPLE, and Xcerpt; secondly the reactive rule query language Algae; thirdly and last the “navigational access” query language Versa. Although subjective, this choice is arguably a good coverage of the diverse language paradigms considered for querying RDF data. It is the authors’ hope and expectation, that this comparison will motivate and trigger further similar studies, thus completing the present article and overcoming its limitation.

1 Introduction

Query Answering on the Semantic Web

Query answering is as central to the Semantic Web as it is to the conventional Web. Indeed, the Web as well as the emerging Semantic Web can be seen as information systems; and query answering is an essential functionality of any information system.

The Semantic Web is a research and development endeavor aiming at overcoming limitations of today's Web. It has has been described as follows by W3C founder Tim Berners-Lee, Jim Hendler, and Ora Lassila:

> "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." [16]

In the Semantic Web, conventional Web data (usually represented in (X)HTML or other XML formats) is enriched by meta-data (represented, e.g., in RDF, Topic Maps, OWL) specifying the "meaning" of other data and allowing Web-based systems to take advantage of "intelligent" reasoning capabilities.

Query answering on the Semantic Web might be seen as more complex than querying on the conventional Web because the "meaning" conveyed by meta-data has to be properly "understood" and processed. In particular, query languages for RDF may convey RDF/S's semantics as expressed, e.g., by RDF type triples.

Focus of this Article

This article is

1. an introduction into query languages for the Semantic Web;
2. an in-depth comparison of the languages introduced along prominent language constructs and concepts.

Only RDF query languages are considered in this article. The reason for this is, that as of the writing of this paper, query languages for other Semantic Web data modeling formalisms, especially OWL, still are an open research issue, and only a very small number of, furthermore incomplete, proposals for querying Semantic Web data modeled after other formalisms than RDF are known.

Furthermore, only a few RDF query languages are considered in this article. This limitation is motivated both by the objective of an in-depth comparison of the languages addressed and by space limitations. During the three years before the writing of this article, more than three dozen proposals for RDF query languages have been published! Not only such a large number, but also the often immature nature of the proposals makes the focus on few, but representative languages a necessary condition for a non-trivial comparison.

In the spirit of a practical introduction into these query languages, we have taken an example-centered approach. We believe that this is advantageous to the reader to quickly gain an impression of the language and constructs. Furthermore, a more formal treatment of the languages is impeded by the lack of (published) formal semantics. In Section 5, however, different semantics for interesting language constructs are addressed and compared in select cases.

This article builds upon and complements the survey [5] of Semantic Web query languages co-authored in 2005 by some of the authors of the present article.[5] While the focus of the 2005 survey has been a complete, but therefore necessarily somewhat shallow coverage of Semantic Web query languages, including on the one hand query languages for Topic Maps and on the other hand all known "dialectal" variations of RDF query languages. In contrast, the present article is focused on an in-depth comparison of a few selected RDF query languages that the authors consider representative. Although building upon the survey [5], this article is self-contained.

At least the first part, of the article is mostly of an introductory nature. We believe, however, that also researchers and scientists already acquainted with RDF query languages can benefit from the presented material. This applies particularly to the comparison of language constructs and evaluation methods in the second part. We hope that the direct comparisons reveal choices that language designers face when deciding which constructs to support in which way, and that language users face when deciding which languages are suitable for their particular needs.

Language Selection and Order

This article aims at introducing from the perspective of the authors interesting and representative selection of query languages proposed for RDF:

- Firstly, the "relational" or "pattern-based" query languages SPARQL, RQL, TRIPLE, and Xcerpt (with its visual "twin" visXcerpt).
- Secondly, the "reactive rule" query language Algae.
- Thirdly, the "navigational access" query language Versa.

Although incomplete and admittedly subjective, this choice can be seen as a good coverage of the diverse language paradigms considered for querying RDF data.

It is the authors' hope and expectation that this comparison will motivate further similar studies that complete the present article and overcome its limitation. It is also the authors' hope that this article will provide Semantic Web practitioners and researchers alike with a good introduction into query answering on the Semantic Web even though it does not address all query languages proposed for the Semantic Web.

Structure of this Article

The following three questions are at the heart of this article and give it its structure:

[5] Sections 2 and 3 are shortend versions of corresponding sections of [5].

1. what are the core *paradigms* of each query language,
2. what *language constructs* do different languages offer to solve tasks such as path traversal, optional selection, or grouping,
3. how are they *realized*?

In Section 2, the RDF/S data model, a running example, the RDF/S semantics and serialization formats are introduced. Section 3 begins by presenting a categorization of Semantic Web queries and sample queries for each category. Subsequently, in Section 4 the RDF query languages selected are introduced—grouped according to their families, i.e., "relational" or "pattern-based", "reactive rule" and "navigational access". For each language considered, some of the sample queries are formulated. For the sake of conciseness and simplicity, not all sample queries are expressed in each language considered. In Section 5 a summary and comparison of language features observable and desirable for RDF query languages is given. Section 6 examines evaluation methods of Semantic Web queries. Section 7 concludes this survey.

2 A Brief Introduction to RDF and RDFS

2.1 Data Model

RDF [10, 59] data are sets of "triples" or "statements" of the form *(Subject, Property, Object)*. RDF data are commonly seen as directed graphs the nodes of which are statement's subjects and objects and the arcs of which correspond to statement's properties, i.e., an arc relates a statement's subject with the statement's object. Properties are also called "predicates". Nodes (i.e., subjects and objects) are either

1. labeled by URIs describing Web resources,
2. or labeled by literals, i.e., scalar data such as strings or numbers,
3. or are unlabeled and called anonymous or "blank nodes".

Blank nodes are commonly used to group or "aggregate" properties. Specific properties are predefined in the RDF and RDFS recommendations [21, 53, 59, 69], e.g., rdf:type for specifying the type of resources, rdfs:subClassOf for specifying class-subclass relationships between subjects/objects, and rdfs:subPropertyOf for specifying property-subproperty relationships between properties. Furthermore, RDFS has "meta-classes", e.g., rdfs:Class, the class of all classes, and rdf:Property, the class of all properties.[6]

RDFS [21] allows one to define so-called "RDF Schemas" or "ontologies", similar to object-oriented data models. The inheritance model of RDFS exhibits the following peculiarities:

1. resources can be classified in different classes that are not related in the class hierarchy,

[6] This survey tries to use self-explanatory prefixes for namespaces where possible.

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. RDF does not describe which properties can be associated with a class, but instead the domain and range of a property.

Based on an RDFS schema, "inference rules" can be specified, for instance the transitivity of the class hierarchy, or the type of an untyped resource that has a property associated with a known domain.

RDF can be *serialized* in various formats, the most frequently used being (RDF/) XML. Early approaches to RDF serialization have raised considerable criticism due to their complexity. As a consequence, a surprisingly large number of RDF serializations have been proposed, cf. [26] for a detailed survey.

2.2 Running Example: Classification-Based Book Recommender

In the following, queries in a simple book recommender system describing various properties and relationships between books are considered as running examples.[7] The recommender system describes properties of 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 1 depicts these data as a (simplified) RDF graph [21, 59, 63]. Note in particular that a Historical_Novel is both, a Novel and an Essay, and that books may optionally have translators, as is the case for *Bellum Civile*.

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. This simple and small ontology is sufficient to illustrate the most important aspects of RDF query languages.

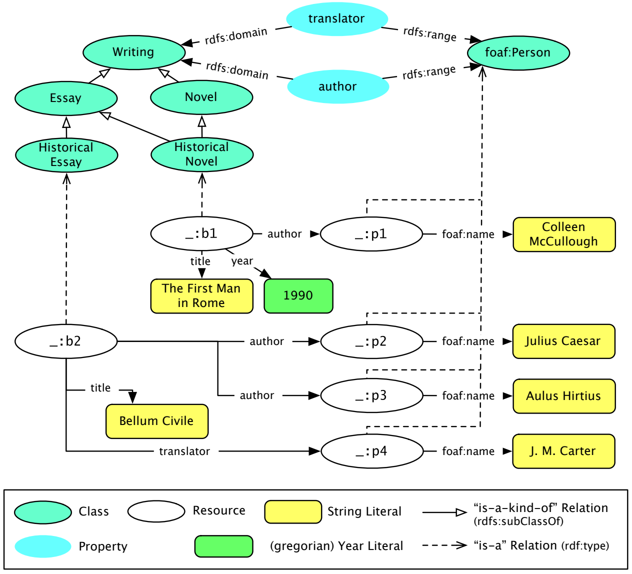
The RDF representation of the sample data refers to the "simple datatypes" of XML Schema [17] for scalar data: Book titles and authors' names are "strings", (untyped or typed as xsd:string), publication years of books are "Gregorian years", xsd:gYear. The sample data are assumed to be accessible at the URI `http://example.org/books#`. Where useful, e.g, when referencing the vocabulary defined in the ontology part of the data, this URL is associated with the prefix books.

Representation of the Sample Data in RDF. The RDF representation of the book recommender system directly corresponds to the simplified RDF graph in Fig. 1. It is given here in the *Turtle* serialization [7].

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
```

[7] The same example is used in [5].

Fig. 1 Sample Data: representation as a (simplified) RDF graph.



```
@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 rdf:Property ;
         rdfs:domain :Writing ;         rdfs:range foaf:Person .
:translator a rdf: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"] .
_:b2     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 "_:".

2.3 Semantics

The meaning of RDF data (e.g., what means "book"?) cannot be fully understood by applications and is interpreted in different ways also by human readers. Naturally, it depends on social, cultural, temporal and other types of context information. However, RDF/S allow to specify part of the semantics of applications (e.g., "a book might have an author").

As is common practice for *declarative* languages, the semantics of RDF/S is specified in terms of a model theory [39, 53]. RDF applications should be able to derive information using the inference rules for basic RDF, while only schema-aware applications are expected to take into account information provided by RDFS inference rules.

3 Sample Queries

The RDF query languages considered in this article are illustrated and illustrated using five different types of queries against the sample data.[8] This categorization is inspired by Maier [67] and Clark [34].

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)"*

Extraction queries extract substructures, and can be considered as a special form of Selection Queries returning not only explicitly queried resources or statements, but entire subgraphs.

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:

Query 3 *"Select all data items except ontology information and translators from the book recommender system."*

Restructuring queries: In Web applications, it is often desirable to *restructure* data, possibly into different formats or serializations. 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:

[8] Again, these queries are mostly the same as in [5].

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 four 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 four statements "X is a statement", "X has subject *Julius Caesar*", "X has predicate *author*", and "X 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 $\mathsf{max}(\cdot)$, $\mathsf{sum}(\cdot)$, $\dots$) and *structural aggregation* working also on structural elements (like "how many nodes"). Query 5 uses the $\mathsf{max}(\cdot)$ 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."*

Combination and inference queries: It is often necessary to *combine* information that is 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.

4 The RDF Query Language Families

In this survey, we focus on three groups of RDF query languages differing in what the authors perceive as central paradigms of the languages:[9] Languages following the relational or pattern-based paradigm use selection constructs similar to selection-projection-join (SPJ) queries. Though they share a common query core, the languages in this group vary quite noticeably, some extending SPJ queries very conservatively, others going well beyond with novel constructs aiming to adequately support the specifics of RDF. The second group is set apart by the use of reactive rules but otherwise shares some commonality with the first group. The final group is more distinctly separated by preferring navigational access and path expressions over patterns.

Figure 2 may serve as orientation through the "language zoo" discussed in this chapter and includes also "dialects" and variants that are only briefly mentioned in the following.

4.1 The Relational Query Languages SPARQL, RQL, TRIPLE, and Xcerpt

The SPARQL Family *SPARQL* [84] is a query language that has already reached candidate recommendation status at the W3C, and is on a good way to become *the* W3C recommendation for RDF querying. It has its roots in *SquishQL* [76] and *RDQL* [91].

Querying RDF data with languages in the SPARQL family amounts to matching graph patterns that are given as sets of triples of subjects, predicates and objects. These triples are usually connected to form graphs by means of joins expressed using several occurrences of the same variable. SPARQL uses the Turtle [7] serialization format for RDF as basis for its own triple syntax. It inherits certain syntactic shorthands from Turtle: e.g., *predicate-object lists* allow several statements to share the same subject without repeating the subject. Pairs of predicates and objects following the subject are separated by colons. *Object lists* are shorthands for several statements sharing both the subject and the predicate, the objects being separated by commas.

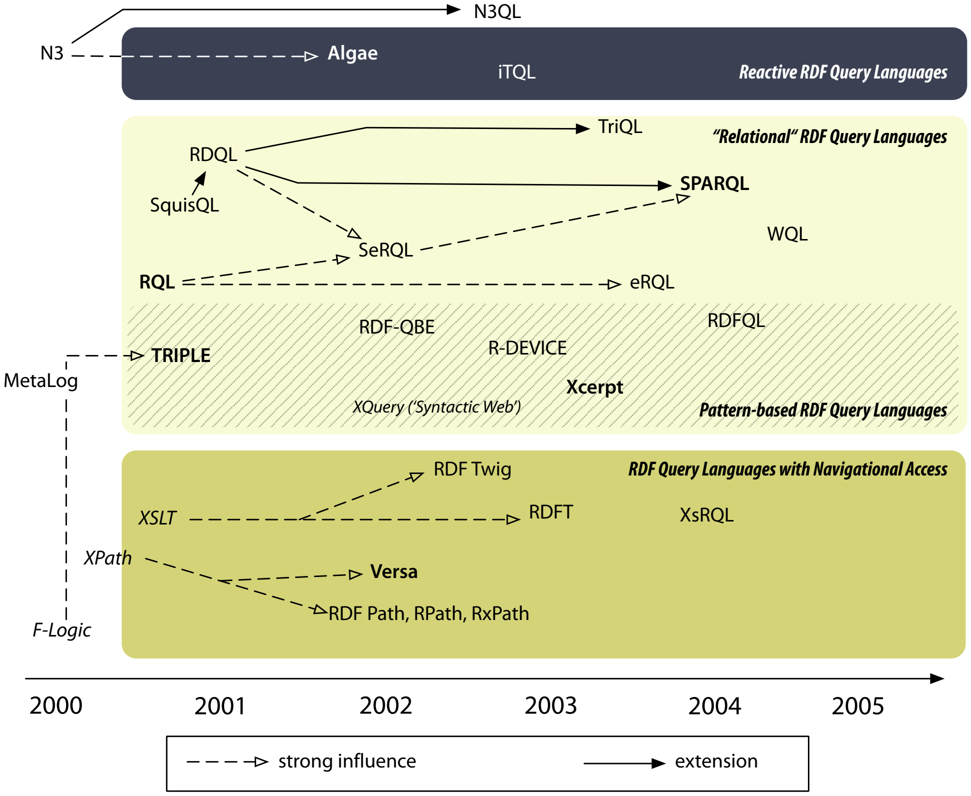
Solutions to SPARQL (or SquishQL or RDQL) queries are given in the form of result sets, for which also an XML format has been specified [9]. In SPARQL, result sets are sets of mappings from the variables occurring within the query to nodes of the queried data. Although RDQL and SquishQL are predecessors of SPARQL, this section presents realizations of the sample queries only in SPARQL. The formulation in the other members of the SPARQL family are very similar though some of the queries use features only recently added and not available in RDQL and SquishQL.

In SPARQL, Query 1 is expressed as follows.

```
PREFIX books: <http://example.org/books#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

[9] See [5] for a more comprehensive survey of Semantic Web query languages.

Fig. 2 Chronological Overview of RDF Query Languages (in **bold** typeface: languages covered in this survey; in *italic* typeface: non-RDF (mostly XML) query languages with proposals/extensions for querying RDF; MetaLog's unique approach to RDF querying based on a natural language interface defies classification in this framework); N3QL is not classified due to incomplete description.



```
SELECT ?essay ?author ?authorName
FROM   <http://example.org/books>
WHERE  { ?essay rdf:type books:Essay .
         ?essay books:author ?author .
         ?author books:name ?authorName . }
```

The WHERE clause specifies the graph pattern to match using variables to select data. Variables are recognized by either ? or $ prefix. Triples are connected to graph patterns using "." (colon). The FROM clause specifies the URL (or some other identifier) of the data to be queried and the SELECT clause the result variables.

Extraction queries like Query 2 can only be approximately expressed in all members of the SPARQL family, because recursive traversals of the data are not possible. Thus one cannot extract all information relevant to a particular resource. Collecting all outgoing edges of a node together with the directly linked objects of these predicates is possible and is showcased in the sample code below. As can be seen, SPARQL does not syntactically differentiate between variables for predicates and for resources, as opposed to RQL discussed below. Also the extraction of information occurring at a fixed distance from the resource representing the book named "Bellum Civile" is possible by adding further statements to the query below.

```
PREFIX books: <http://example.org/books#>
SELECT ?property ?propertyValue
FROM   <http://example.org/books>
WHERE  {?essay books:title "Bellum Civile" .
        ?essay ?property ?propertyValue . }
```

Another way to approximate extraction queries are SPARQL's DESCRIBE queries that allow the retrieval of "descriptions" for resources. The exact extent of such a "description" is not defined in [84], but concise bounded descriptions [96] are referenced as a reasonable choice. These represent a form of predefined extraction query that returns all immediate properties for a resource as well as the immediate properties of all blank nodes that are reachable from the resource to be described without other named resources in between.

The FILTER keyword is used in SPARQL to eliminate result sets which evaluate to false when substituted in the boolean expressions given in the body of the FILTER clause. A query that finds the persons that have authored a book with title "Bellum Civile" can be expressed in SPARQL as follows:

```
PREFIX books: <http://example.org/books#>
SELECT ?person
FROM   <http://example.org/books>
WHERE  { ?book books:author ?person .
         ?book books:title ?title .
         FILTER (?title = ’Bellum Civile’) }
```

The three queries mentioned above are also expressible in SPARQL's predecessors SquishQL and RDQL with a slightly different syntax but almost identical structure. SPARQL and its relatives do not support RDF/S inferencing, which means that among other tasks, querying all resources of type books:Writing of the example data above would not return any results, because there are no resources which are directly associated with books:Writing via an rdf:type property. If the SPARQL family provided support for inferencing, the resources represented by the blank nodes _:b1 and _:b2 in the serialization in Section 2.2 could be returned as results to the query in compliance with the rule RDFS9 of the RDFS semantics. One can argue that RDF/S and OWL reasoning should not be a task of the query language, but should be provided by an underlying black box reasoner. Given such a reasoner that transparently provides the full RDFS entailment graph, i.e., the closure graph under the RDF/S inference rules, the languages of the SPARQL family can very well answer queries such as the one just mentioned.

There are several other characteristics and also limitations of the members of the SPARQL family, which deserve to be mentioned:

– Queries cannot be composed or nested.
– Negation can only be used in FILTER clauses (they are called AND-clauses in SquishQL and RDQL), but not in WHERE clauses, i.e., triple patterns can only occur positively.
– Due to the lack of recursion, members of the SPARQL family cannot express certain kinds of inference queries such as 8 and extraction queries (as has been mentioned above).

SPARQL being a descendant of RDQL and SquishQL, it provides some additional features, that go beyond the queries mentioned above and which are not included in RDQL and SquishQL. Among these new features are:

– The construction, using CONSTRUCT clauses, of new RDF graphs with data from the RDF graph queried. Just as the query patterns, the construct patterns are specified as sets of triples with variables serving as placeholders. Naturally, all variables appearing within the construct pattern must also appear within the query.
– The possiblity to return, using DESCRIBE clauses, "descriptions" of the resources matching the query part. The exact meaning of "description" is left undefined, cf. [96] for a proposal.
– The specification of OPTIONAL triple or graph query patterns, i.e., data that should contribute to an answer if present in the queried data, but whose absence does not prevent from returning an answer. A corollary of is the ability of SPARQL to test for absence of triples (i.e., negation-as-failure). E.g., finding all books which do not have a translator is achieved by using the OPTIONAL keyword and a FILTER expression requiring that the optional variable is *not* bound included in the optional query part:

```
PREFIX   books: <http://example.org/books#>
PREFIX   rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT   ?writing
FROM     <http://example.org/books>
WHERE    { ?writing books:author _:Author .
           OPTIONAL { ?writing books:translator ?translator } .
           FILTER (!bound(?translator)) }
```

- The expression of disjunctions of queries with the keyword UNION.
- Querying named graphs. First introduced in TriQL [18], another variant of RDQL, named graphs allow the scoping of triples and triple patterns: A query is evaluated not against a single set of triples but rather against a set of such sets each associated with a name (in form of a URI). The FROM NAMED clause can limit the matching of the triple pattern in the associated WHERE to the graphs with the specified names.

In contrast to other RDF query languages, SPARQL supports four different *query result forms*, which vary in the type of results returned. Only queries formulated using CONSTRUCT or DESCRIBE are closed in the sense that the results are RDF graphs just as the queried data. Queries using ASK return a boolean value and is used to find out whether a query pattern matches with the data. The SELECT query pattern is used to collect variable bindings from query patterns just as in SquishQL and RDQL.

The CONSTRUCT clause provides a straightforward enhancement over mere collection of variable bindings. Following the CONSTRUCT keyword, a result template is specified, which is an RDF graph that contains some or all of the variables from the query pattern in the WHERE-clause. For each match of the query pattern with the queried data, the result template is filled with the corresponding variable bindings, and the resulting RDF graph is included in the answer graph. However, CONSTRUCT patterns are rather limited missing, e.g., any ability for grouping (and thus can not construct new RDF containers or collections).

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 .
            FILTER (?x != ?y) }
```

One of SPARQL's design principles is that queries should be easily derivable from RDF graphs. Thus, any RDF graph can be included in the WHERE-clause of a SPARQL query in Turtle [7] syntax. A further result of this design principle is that blank nodes are allowed to appear within query patterns. It must be emphasized that blank nodes in query patterns are not required to match with blank nodes of the data to be queried, but are mere syntactical sugar for existentially quantified variables.[10]

Besides query result forms, SPARQL provides the *solution modifiers* DISTINCT, ORDER BY, LIMIT, and OFFSET. DISTINCT eliminates duplicates in the sets of variable bindings, LIMIT specifies an upper bound for the number of solutions, OFFSET is used to omit the first n solutions of the solution sequence, and ORDER BY allows to order the solution sequence ascending or descending according to one or more variable bindings or according to a function.

[84] contains a formal semantics for SPARQL. For details on SPARQL's semantics refer to [84] and to the tutorial on SPARQL in this volume [81]. The latter, in particular, motivates the, at a first glance, slightly odd definition of SPARQL's semantics.

The RQL Family Under "RQL family", we group the languages *RQL* [57] and *SeRQL* [22]. Common to these languages is that they support combining data and schema querying. In the case of RQL, the RDF data model deviates slightly 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 rdf:Property the class of all properties. They make possible a flexible type system tailored to the specificities of RDF and RDFS.

In the following discussion we concentrate on **RQL**, the "RDF Query Language", that has been developed at ICS-FORTH [31, 54, 55, 56, 57]. Its most distinguishing feature is a strong support for typing as well as a more complete set of advanced language operators such as set operations, aggregation, container construction and access than in most other RDF query languages.

SeRQL aims to be a more accessible derivate of RQL. Therefore several syntactic shorthands (e.g., object-property and object lists and optional expressions, all three later adopted in SPARQL) are introduced for common query situations. Also SeRQL drops built-in support for typing beyond literals, presumably under the impression that the multitude of language constructs provided in RQL makes the language too complex. The same reasoning applies for advanced query constructs such as set operations, universal quantification, aggregations, etc.

Another derivate of RQL is eRQL, a radical simplification of RQL based mostly on a keyword-based interface. It is the expressed goal of the authors of eRQL to provide a *"*Google-like *query language but also with the capacity to profit of the additional information given by the RDF data"*.[11] The resulting language is, unsurprisingly, of rather limited expressiveness and can not express most of the sample queries.

[10] See http://lists.w3.org/Archives/Public/public-rdf-dawg-comments/2006Jan/0073-.html for a discussion about blank nodes in SPARQL queries.

[11] `http://www.dbis.informatik.uni-frankfurt.de/~tolle/RDF/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[12]. A similar query, using subPropertyOf instead of subClassOf, returns the sub-properties of a property. The following three queries returns the domain ($C1) and range ($C2) of the property author defined at the URI named books. The prefix $ indicates "class variable", i.e., a variable ranging on schema classes. It can be expressed in RQL in three different manners:

1. using class variables:

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

2. using a *type constraint*:

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

3. without class variables or type constraints:

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

While the first two queries return exactly the same result—namely the domain and range of the books:author-property and all possible combinations of their subclasses—the third query does not include the domain and range of books:author itself but only the combinations of their subclasses. There is another subtle difference: the first two queries should only return class combinations for which actual statements exist, the third should also return class combination where no actual statement for that combination exists.

The query topclass(books:Historical_Essay) returns the top of the subsumption hierarchy, i.e., books:Writing, cf. Figure 1. Similar constructs for querying the leaves of the subsumption hierarchy or the nearest common ancestor of the two classes are available. Moreover, RQL has "property variables" that are prefixed by @ and which can be used to query RDF properties (just as classes can be queried using class variables). The following query, with property variables prefixed by @ 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}.

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

Data queries. With RQL, data can be retrieved by its types or 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, i.e., inferred) extent. The query `books:Writing` returns the resources classified as books:Writing or as one of its sub-classes. This query can also be expressed as follows: `SELECT X FROM books:Writing{X}`. Prefixing the variable `X` with ˆ in the previous queries, yields queries returning only resources directly classified as books:Writing, i.e., for which a statement $(X,$ rdf:type, books:Writing$)$ exists. The extent of a property can be similarly retrieved. The query `^books:author` returns the pairs of resources X, Y that are in the books:author relation, i.e., for which a statement $(X,$ books:author, $Y)$ exists. RQL offers extended dot notation as used in OQL [29], 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#
```

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}` finds the classes under which the resource `X` is directly classified.

Further features of RQL are not discussed here, e.g., support for containers, aggregation, and schema discovery. Although RQL has no concept of "view", its extension RVL [66] gives a facility for specifying views.

RQL has been criticized 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. On the other hand, RQL is far more expressive than most other RDF query languages, especially those of the SPARQL family. Most queries of Section 3, except those queries referring to the transitive closures of arbitrary relations, can be expressed in RQL.

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 RQL's extension 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#
```

A formal semantics for RQL has been defined together with the language, e.g., in [57].

TRIPLE [51, 92, 93] is a rule-based query, inference, and transformation language for RDF. TRIPLE is based upon ideas published in [40]. TRIPLE's syntax is close to F-Logic [58]. F-Logic is convenient for querying semi-structured data, e.g., XML and RDF, as it facilitates describing schema-less or irregular data [64]. Other approaches to querying XML and/or RDF based on F-Logic are XPathLog [75] and the ontology management platform Ontobroker[13]. 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.

[13] http://www.ontoprise.de/products/ontobroker

Instead of predefined RDFS-related language constructs, TRIPLE offers Horn logic rules (in F-Logic syntax) [58]. 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 [60, 61], the authors of TRIPLE claim in [93] that TRIPLE is well-suited to query non-RDF meta-data. This can be questioned, especially if, in spite of [44], one considers the rather awkward mappings of Topic Maps into RDF proposed so far.

TRIPLE differs from Horn logic and Logic Programming as follows [93]:

- 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 [43] can be used for traversing several properties.
- TRIPLE provides concise support for reified statements. Reified statements are expressed in TRIPLE enclosed in 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. [93] gives the following implementation of a part of RDFS's semantics:

```
rdf     := 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'.
rdfs    := 'http://www.w3.org/2000/01/rdf-schema#'.
type    := rdf:type.
subPropertyOf := rdfs:subPropertyOf.
subClassOf    := rdfs:subClassOf.

FORALL Mdl @rdfschema(Mdl) {
  transitive(subPropertyOf).
  transitive(subClassOf).
  FORALL O,P,V O[P->V] <-
               O[P->V]@Mdl.
  FORALL O,P,V O[P->V] <-
               EXISTS S S[subPropertyOf->P] AND O[S->V].
  FORALL O,P,V O[P->V] <-
               transitive(P) AND EXISTS W (O[P->W] AND W[P->V]).
  FORALL O,T   O[type->T] <-
               EXISTS S (S[subClassOf->T] AND O[type->S]).
}
```

Inference from range and domain restrictions of properties are not implemented by the rule given above. This is no limitation of TRIPLE, though, as they can be realized by the following additional rules:

```
  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 rules implementing RDFS' semantics. Most queries of Section 3 can be expressed in TRIPLE. Aggregation queries cannot be expressed in TRIPLE, for the language does not support aggregation.

[93] 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 blank nodes, are not supported by TRIPLE.

Xcerpt. Xcerpt [13, 24, 88, 89], 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 data). Therefore the approach of querying an XML serialization of Semantic Web data is feasible in Xcerpt, but it is not as natural as directly querying the RDF data. Xcerpt uses common language constructs for querying data in several different formats and is therefore very useful for authoring applications that combine all kinds of Web data. This survey focuses on applying Xcerpt to querying RDF data, but querying XML and Topic Maps with Xcerpt is quite similar (cf. [5]).

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 3 can be expressed in Xcerpt. The following Xcerpt programs show solutions for the queries against the RDF serialization from Section 2.

[19] 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#"
        ns-prefix books = "http://example.org/books#"

GOAL
  result [
    all essay [
      id [ var Essay ],
      all author [
        id [ var Author ],
        all name [ var AuthorName ]
      ] ] ]
FROM
  and(
    RDFS-TRIPLE [
      var Essay:uri{}, "rdf:type":uri{}, "books:Essay":uri{} ],
    RDF-TRIPLE [
      var Essay:uri{}, "books:author":uri{}, var Author:uri{} ],
    RDF-TRIPLE [
      var Author:uri{}, "books:authorName":uri{}, var AuthorName ] )
END
```

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 [53] (cf. [19] 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 [86] 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#"
        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 just like 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* (possibly containing cycles) 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.

Xcerpt rules are convenient for making the language "RDF serialization transparent". For each RDF serialization, a set of rules expresses a translation from or into that serialization. However, the rules for parsing RDF/XML [10], the official XML serialization, are very complex and lengthy due to the high degree of flexibility RDF/XML allows. They can be found in [19], similar functions for parsing RDF/XML in XQuery are described in [87]. The following rules parse RDF data serialized in the RXR (Regular XML RDF) format [4], a far simpler and more regular RDF serialization.

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 [19].

```
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 as rxr:subject{{}},
        rxr:predicate{ attributes{ rxr:uri{ var Predicate } } },
        var O as 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 [19].

A visual language, called *visXcerpt* [11, 12], 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.

A formal semantics of Xcerpt has been published in [88]. Static type checking methods have been developed for Xcerpt [25, 98] that are based on seeing tree grammars in their various disguises, e.g., DTD, XML Schema, RelaxNG, as definitions of abstract data type. Recent work [28, 90] on Xcerpt focuses on efficient evaluation of Xcerpt's high-level constructs.

There is quite a number of other query languages that fall into this group but can not be covered here for space reasons (for further details see [5]). Further investigaton of such languages might start with R-DEVICE [6], RDF-QBE [85], and RDFQL [1].

4.2 The Reactive Rule Query Language Algae

Algae[14] 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[15]-like rules. (2) Answers to Algae queries are bindings for query variables as well as triples from the RDF graph as "proofs" of the answer. Algae queries can be composed. Syntactically, Algae is based on the RDF syntax N-triples [46], a subset of the N3 [14] notation for RDF. This subset excludes specifically N3 rules or queries as used in the N3QL proposal [15]. 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 )
```

[14] Also called "Algae2". This survey follows [83] and retains the name "Algae".

[15] ECA stands for event-condition-action.

?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

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’ [82].

No formal semantics has been published for Algae.

Algae is not the only RDF query language that provides reactive rules: iTQL [2] is used in the Kowari Metastore and provides querying, update, and transaction management functionality, for details see [5]. iTQL is also one of the few RDF query languages with a form of unrestricted closure path expressions (thanks to the trans function). RUL [65], the RDF update language, provides update expressions on top of RQL.

4.3 The Navigational Access Query Language Versa

Developed as part of the Python-based *4Suite* XML and RDF toolkit[16], **Versa** [77, 78, 79] is a query language for RDF inspired, but significantly different from XPath[33, 45]. 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

[16] http://4suite.org/

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 evaluated 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 titled "Bellum Civile" are returned by

```
(books:Essay <- rdf:type - *) |- books:title -> eq("Bellum Gallicum").
```

Versa's function traverse serves to traverse paths of arbitrary length, e.g., the following query returns all sub-classes of books:Writing:

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

Similarly, Versa's function filter provides a general filter, e.g., all essays entitled "Bellum Gallicum" having a translator named "J. M. Carter" are returned by the following query:

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

Selection and extraction queries can be easily implemented in Versa, although the selection of related items is not very convenient, as the above implementation of Query 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(rdf: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 -> *))"
        )
```

No formal semantics has been published for Versa.

Aside from Versa, most RDF query languages that fall into this group are derivatives of XPath or XSLT or are at least very similar to these XML query languages, for details once more refer to [5]. There are a few proposals for XPath-style RDF path languages (RDF Path [80], RPath [74], RxPath [94]), however all proposals are very limited in expressiveness and often immature. [86, 87] suggests the use of XQuery for querying RDF, TreeHugger [95] and RDF Twig [97] do the same for XSLT (1.0), the latter two relying on external functions for preprocessing the RDF data. RDFT [38] suggests an RDF template language in the style of XSLT, as does [62]. Both approaches seem to have been abandoned.

This section has introduced a number of RDF query languages divided in three groups. For an overview of the discussed languages and their relations, refer again to Figure 2. The following two sections relate the introduced languages comparing their approaches to selection, construction, evaluation, etc.

5 Language Constructs Compared

The previous section establishes a basic understanding of interesting exemplars of RDF query languages. This broad overview of languages is complemented in this section with a close look at specific language concepts and constructs. For instance, selecting optional data is essential for RDF, since all properties are optional by default. However, different languages provide quite different means to handle such data. All these language constructs are compared over several of the languages from the previous section as appropriate to show the range of solutions for the particular need.

For the purpose of this section, the constructs are divided in three classes: selection, construction, and procedural abstraction or view definition.

5.1 Selection

The basic functionality of any query language is selection, i.e., the ability to characterize subsets of the queried data that match the user's query intent. In relational databases where the schema of the data is well-known, such characterizations are often based on few attributes of the sought-for data items and possibly a small number of relations with other data items. On semi-structured data such as XML or RDF, selection becomes more centered around the position of the sought-for data items within the structure of the queried data. Some RDF (and most XML) query languages therefore provide not just selection based on attribute value, but richer selection constructs.

Triple Patterns vs. Path expressions

Triple patterns. The basic form of selection construct is a triple pattern that corresponds to a relational selection-(projection-)join query. A triple pattern consists of a conjunction of one or more triples, that are just like data triples but may additionally be extended with query constructs such as variables. SPARQL uses triple patterns in Turtle syntax. E.g.,

```
?essay books:title "Bellum Civile"
```

selects the resources with "Bellum Civile" as value of the books:title property. This basic form of a triple pattern is like a selection operation from the relational algebra. If variables occur in several triples in the same triple pattern, that pattern becomes a selection-(projection-)join query[17], e.g.,

```
?essay books:author ?author.
?author foaf:name "Julius Caesar"
```

Joins expressed, e.g., through multiple occurrences of the same variable in the same pattern query are even more prevalent in RDF than in usual relational queries. This is partially due to the binary nature of RDF properties. Furthermore, one often needs to "traverse" several intermediary nodes in the RDF graph to select the actually used data items.

Specifying such traversals in a succinct way has been considered not only in the context of RDF, but also in the context of relational (GEM [100]), object-oriented ([43]) and XML ([33]) data. The most successful and for semi-structured and XML query languages widely accepted construct for specifying structure

[17] Triple pattern queries as discussed here and used, e.g., in SPARQL have more or less the same expressiveness and evaluation complexity as relational SPJ-queries.

traversal are **path expressions**. Essentially, they allow the omission of variables for intermediary nodes that are just used to "reach" the target nodes. E.g., the above SPARQL query can also be written as

```
?essay books:author [foaf:name "Julius Caesar"].
```

which uses the ability of SPARQL's syntax to omit blank nodes (i.e., existentially quantified variables) in queries and is tantamount to a path expression. RQL specifically introduces path expressions with a syntax similar to OQL's dot notation:

```
{Essay}books:author.foaf:name{A}.
```

Path Expressions. Path expressions constructs can be classified along their intended use and expressiveness in three classes:

1. *Basic path expressions* are only abbreviations for triple patterns as seen in SPARQL or RQL. They allow only the specification of fixed length traversals, i.e., the traversed path in the *data* is of same length as the path expression. These path expressions are not more expressive than triple patterns (and therefore SPJ queries), but are nevertheless encountered in several query languages as "syntactic sugar". Examples of query languages with only basic path expressions are GEM [100], OQL [29], SPARQL [84], and RQL [84].
2. *Unrestricted closure path expressions* are a common class of path expressions that adds to the basic path expressions the ability to traverse arbitrary-length paths. XPath path expressions (disregarding XPath predicates for the moment) fall into this category with closure axes such as descendant. This type of path expressions is very common in XML query languages (e.g., XML-QL [41], Quilt [30], XPath and all XML query languages based on XPath). It is also used in the RDF query language iTQL[2]. Its expressiveness is indeed higher than that of basic triple patterns (SPJ queries). It can be realized in languages that provide only triple patterns but additionally (at least linear) recursive views. SQL-99 is an example of a language that provides no closure path expressions but linear recursion and thus can emulate (unrestricted) closure path expressions. For RDF, there are few query languages that fall into this class since RDF has, in contrast to XML, no dominating hierarchical relation but many relations of equal importance. This makes unrestricted closure often too unrestrictive for interesting queries.
3. Therefore, several RDF query languages provide *generalized or regular path expressions*. Here, full regular expression syntax including repetition and alternative is provided on top of path expressions. E.g., `a*.((b|c).e)+` traverses all paths of arbitrary many a properties followed by at least one repetition of either a b or a c in each case followed by an e. Such regular path expressions are provided, e.g., by Versa's traverse operator, Xcerpt's qualified descendant, or the XPath extension with conditional axes [71]. The latter work showed that regular path expressions are even more expressive than unrestricted closure path expressions and a path language like XPath becomes indeed first-order complete with the addition of regular path expressions.

> Nevertheless, direct language support is not only justified by the ease of use for the query author but also by complexity results, e.g., in [70] that show that regular path expressions do not affect the complexity of a query language such as XPath and can be evaluated in polynomial time w.r.t. data and query size. Simulation of regular path expressions using triple patterns (SPJ queries) and recursive views is possible but the resulting queries become excruciatingly complex even for simple regular path expressions.

Summarizing, path expressions provide convenient means to specify structural constraints in RDF queries and are therefore supported by a large number of RDF query languages. However, surprisingly many RDF query languages ignore (unrestricted or regular) closure path expressions. This is surprising as these path expressions make query authoring (they allow avoiding recursive views) easier and can be implemented efficiently as research on these constructs for XML query languages has shown. In particular, unrestricted closure path expressions can be implemented nearly as efficiently as basic path expressions using, e.g., tree labeling schemes [48] or closure indices.

Closure Subgraph Extraction Closely related to (regular or unrestricted) closure path expressions, is the issue of subgraph extraction: Since schema and extent of RDF data are often, at best, only vaguely known, extracting interesting portions of the data whose extent is not known statically (i.e., at query authoring or compilation time) becomes an often encountered problem: E.g., given information about authors and books, extract all information on one book, e.g., for export into a bibliography management application or for styled display on a Web site.

It should be immediately clear, that closure subgraph extraction is easily achieved in languages providing (regular or unrestricted) closure path expressions. Regular path expressions are probably needed in the case of RDF to define a reasonable subgraph, e.g., by traversing only certain relations, traversing only a certain number of times, or stopping at resources with certain characteristics.

What about languages with only triple patterns and/or basic path expressions such as SPARQL, RQL, or RDQL? Some of these languages, e.g., RQL, provide built-in closure for certain fixed, predefined relations, cf. Section 5.1. SPARQL provides one specialized language construct, DESCRIBE, that is intended to return relevant and representative information about resources, e.g., in the style of concise bounded descriptions [96] where a resource is described by its immediate properties and the immediate properties of all blank nodes reachable from the resource without other named resources in between. The intuition here is that further information about the latter blank nodes can not be retrieved in further queries to the RDF data as they are not addressable from outside. The SPARQL specifications, however, does not require DESCRIBE to return concise bounded descriptions but leaves the extent of the returned information up to the implementation. Nevertheless, DESCRIBE is the only construct in SPARQL that approximates closure subgraph extraction.

Schema-aware Selection The discussion of closure path expressions could not be complete without looking at one common way of reducing closure path expressions to basic expressions: It is assumed that closure is only relevant for a few, predefined relations such as rdfs:subClassOf which are known to be transitive. For these, the implementation transparently provides the closure.

This is just one of the effects when RDF query languages provide schema-aware (in this case RDFS-aware) selection. An RDF query language may elect to match the query not against the bare data graph but against the entailment graph according to some set of entailment rules, e.g., the RDFS entailment rules. E.g., RQL provides support for the specific entailment rules of RDFS with some exceptions (acyclic subsumption hierarchy, only part of the axiomatic triples are included). The latter exception is, in fact, needed to guarantee that query answer are always finite, as the RDFS entailment rules in [53] include one axiomatic triple for each integer i to handle rdf:_i properties. Query languages must, in this case, opt for a reasonable restriction, e.g., to include only axiomatic triples for integers $i \leq m$ with m the maximum size of a container in the data.

TRIPLE [93] takes schema-aware querying a step further by providing means to parameterize a query with a "model" containing the rules to use for computing the entailment graph against which the query is to be matched. This allows the treatment of different schema languages in the same query framework.

Similarly, schema-awareness can be achieved in any RDF query language with (recursive) views by providing a collection of rules implementing the schema entailment rules. Xcerpt chooses this approach, as it makes schema access transparent for the query author. However, languages like Xcerpt and Versa that provide regular path expressions allow the query author also to specify queries with adhoc schema-awareness in the queries, e.g., by using a closure path expression like `(rdfs:subClassOf)+` instead of just `rdfs:subClassOf`.

None of these approaches forces the entailment graph ever to be materialized. Rather, it may be lazily (i.e., in a goal-driven backward-chaining manner) computed, partially materialized, or fully materialized depending on the needs of the implementation and the query.

Optional Selection and Disjunctions So far, we have considered pure conjunctive queries only. Disjunction or equivalent union constructs allow the query author to collect data items with different characteristics in one query. E.g., to find "colleagues" of a researcher from an RDF graph containing bibliography and conference information, one might choose to select co-authors, as well as co-editors, and members in the same program committee. On RDF data, disjunctive queries are far more common place than on relational data since all RDF properties are by default optional. Many queries have a core of properties that have to be defined for the sought-for data items but also include additional properties (often labeling properties or properties relating the data items to "further" information such as Web sites) that should be reported if they are defined for the sought-for data items but that may also be absent. E.g., the following SPARQL query returns pairs of books and translators for books that have translators and just books otherwise. If one considers the results of a query as a table with null values, the translator column is null in the latter case.

```
SELECT   ?writing, ?translator
WHERE    { ?writing a books:Essay .
         OPTIONAL { ?writing books:translator ?translator } }
```

Such optional selection eases the burden both on the query author and the query processor considerably in contrast to a disjunctive or union query which has to duplicate the non-optional part:

```
SELECT   ?writing, ?translator
WHERE    { ?writing a books:Essay .
           ?writing books:translator ?translator }
         UNION
         { ?writing a books:Essay }
```

Furthermore, the latter is not actually equivalent as it returns also for writings X with translators one result tuple $(X, \texttt{null})$. Indeed, this points to the question of the precise semantics of an optional selection operator. One can observe that the answer to this question is not the same for different RDF (or XML) query languages. The main difference between the offered semantics in languages such as SPARQL, Xcerpt, or XQuery lies in the treatment of multiple optional query parts with dependencies. E.g., in the expression $A \wedge \mathsf{optional}(B) \wedge \mathsf{optional}(C)$ the same variable V may occur in both B and C. In this case, if we just go forward and use the B part to determine bindings for V those bindings may be incompatible with C, i.e., prevent the matching of C. The way this case of multiple interdependent optionals is handled allows to differentiate the following four semantics for optional selection constructs:

1. *Independent optionals:* Interdependencies between optional clauses is disregarded by imposing some order on the evaluation of optional clauses. SPARQL, e.g., uses the order of optional clauses in the query: The following query selects essays together with translators and, if that translator is also an author, also the author name.

   ```
   SELECT   ?writing, ?person, ?name
   WHERE    { ?writing a books:Essay .
            OPTIONAL { ?writing books:translator ?person }
            OPTIONAL { ?writing books:author ?person .
                       ?person foaf:name ?name } }
   ```

   If we change the order of the two optional parts, the semantics of the query changes: select all essays together with authors and author names (if there are any). The second optional becomes superfluous, as it only checks whether the binding of ?person is also a translator of the same essay but whether the check fails does not affect the outcome of the query.

   It should be obvious that this semantics for interdependent optionals is equivalent to allowing only a single optional clause per conjunction that may in turn contain other optional clauses. Therefore, the above query could also be written as follows:

```
SELECT   ?writing, ?person, ?name
WHERE    { ?writing a books:Essay .
         OPTIONAL { ?writing books:translator ?person
            OPTIONAL { ?writing books:author ?person .
                       ?person foaf:name ?name }
         } }
```

This observation, however, only applies if the optional clauses are interdependent. If they are not interdependent multiple optional clauses in the same conjunction differ from the case where they are nested.

Algae seems to employ the same optional semantics as SPARQL, though the language specification is rather vague at that point.

2. *Maximized optionals:* Another form of optional semantics considers any order of optionals: In the example it would return the union of the orders, i.e., either first binding translators than checking whether they are also authors or first binding authors and author names then checking whether they are also translators. This is more involved than the above form and assigns different semantics to adjunct optionals vs. nested optionals. The advantage of this semantics is that it is equivalent to a rewriting of optional to disjunctions with negated clauses: $A \wedge \mathsf{optional}(B) \wedge \mathsf{optional}(C)$ is equivalent to $(A \wedge \mathsf{not}(B) \wedge \mathsf{not}(C)) \vee (A \wedge \mathsf{not}(B) \wedge C) \vee (A \wedge B \wedge \mathsf{not}(C) \vee (A \wedge B \wedge C)$. This semantics ensures that the maximal number of optionals for a certain (partial) variable assignment is used. This semantics has been introduces in Xcerpt.
3. *All-or-nothing optional:* A rare case of optional semantics is the "all-or-nothing" semantics where either all optional clauses are consistent with a certain variable assignment or all optional variables are left unbound. This semantics can be achieved in SPARQL and Xcerpt using a single optional clause instead of multiple independent ones.

RDF Specificities Following the look at general issues for query languages in the specific context of RDF, this section closes the discussion of selection constructs with a consideration of selection constructs for RDF specificities such as blank nodes, collections, reified statements etc. RDF query languages should support these specificities in some way (possibly only as syntactic sugar) to be considered adequate to the RDF data model.

Blank Nodes. Among the considered specificities, blank nodes are the only ones that introduce new challenges for the query language. For matching, blank nodes are just like any other resource, but obviously do not match if a URI is specified in the query. However, for result construction blank nodes have to be considered specifically, see Section 5.2.

Collections and Containers are RDF's constructs to represent sets, sequences, and similar structures. The difference between containers and collections lies in the fact that containers are always open (i.e., new members may be added through additional RDF statements) and collections may be closed. Both containers and collections are merely vocabulary and representational conventions but do not extend the data model. I.e., a sequence container $\langle A, B, C\rangle$ is reduced to the triples

```
_:1 rdf:type rdf:Sequence
_:1 rdf:_1   A
_:1 rdf:_2   B
_:1 rdf:_3   C
```

Similarly, collections are reduced to binary relations of rdf:first and rdf:last:

```
_:1 rdf:first A
_:1 rdf:rest  _:2
_:2 rdf:first B
_:2 rdf:rest  _:3
_:3 rdf:first C
_:3 rdf:rest rdf:nil
```

However, these reductions result in lengthy and hard to understand triple patterns. Furthermore, querying directly on these representations proves challenging in many RDF query languages. Consider the simple query intent for selecting all members of a container or collection C. This query cannot be expressed in most RDF query languages if C is a collection, as it requires an arbitrary-length traversal of rdf:first and rdf:last edges (or direct support of collections) neither of which most RDF query languages provide including SPARQL. In languages with regular path expressions such as Versa or Xcerpt this query can be expressed as C `rdf:first.(rdf:rest.rdf:first)*` R with R selecting the contained resources. In the case of containers, an RDF query language either needs direct support or some support for regular expressions over property URIs. SPARQL, e.g., can express the query as

```
SELECT   ?contained_resource
WHERE    { ?C ?P ?contained_resource .
          FILTER(regex(str(?P),
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#_\d+")) }
```

where the regular expression `\d+` stands for one or more digits.

RQL is one of the few RDF query languages that provide specific constructs for querying membership in containers and even position in ordered containers. E.g., the above query can simply be expressed as R in C, selecting all resources R in the container C. Though RQL does not yet consider collections, this addition should be straightforward.

Reification. Reified statements are another example for a modeling construct that is reduced to several triples but is often convenient to query without requiring the author to perform the reduction by hand. Indeed, some RDF query languages such as SeRQL [22] and TRIPLE [93] provide specific syntax for reified statements, that allows reified statements to be queried with the same syntax as normal statements. SeRQL simply encloses a triple pattern in curly braces to indicate reification.

5.2 Construction

Where the previous section has focused on how RDF query languages select data from the underlying RDF graph, this section looks at the reporting of the selected data including construction of new data.

Graph Construction vs. Selection-only Surprisingly many RDF query languages are not closed, i.e., their result is not again RDF but often simply sets or sequences of tuples representing alternative variable assignments. Examples of such languages are RDQL [91] and Versa. SPARQL provides both just variable assignments using the SELECT keyword and some limited form of graph construction using the CONSTRUCT keyword which, however, falls short of even the most simple grouping tasks.

Even when considering only variable selection **blank nodes** in results are an interesting challenge for RDF query languages. Blank nodes can not be identified from outside thus any "internal" identifier for a blank node returned as part of a result provides at best existential information (i.e., there is a node that fulfills a query). This makes grouping and aggregation even more important than in relational queries. All the more surprising is the lackluster support for these well-established language features in RDF query languages. RQL is one of the few languages providing aggregation including grouping by sub-queries: The following query selects all resources authored by "Julius Caesar" together with the count of their properties.

```
SELECT R, count(SELECT @P FROM {R @P }
FROM  {R}books:author{A}
WHERE  A = "Julius Caesar"
```

The languages in the SPARQL family mostly lack any form of aggregation thus requiring, e.g., post-processing of query results to solve such queries.

Graph Construction A basic requirement for any query language is closure, i.e., the ability to construct data in the same data model as the queried data. In the case of RDF query languages, quite a number of languages focus on selection only, e.g., Versa and RDQL. Others, such as SPARQL provide graph construction but only the most basic form. Most notably, SPARQL omits any form of **grouping** which severely limits the sort of graphs that can be constructed.

The basic form of graph construction in SPARQL is

```
CONSTRUCT { ?R ?P ?O }
WHERE     { ?R books:author "Julius Caesar". ?R ?P ?O }
```

Constructing a graph with one triple for each property of all resources with author "Julius Caesar". Indeed, SPARQL's constructions are just triple patterns again generating one instance of the triple pattern for each variable assignment produced by the query.

In particular, this means that blank nodes in construct patterns are instantiated once for each variable assignment. There is no way that triples for different variable assignments "share" blank nodes.

Collections and containers. This separate handling of constructed instances prevents any form of **grouping** including the construction of **containers** and **collections**, for both of which some form of grouping is needed. Thus, it is impossible to answer simple queries such as “put the names of hotels for each city in a container/collection” or link each city and all its inhabitants to a common (blank) node. What SPARQL lacks is a proper “identity invention” facility, cf. [3].

RQL provides specialized constructs for constructing collections and containers and allows arbitrary grouping using nested queries, but also lacks proper treatment of blank nodes in construction.

Minimal Result Graphs. In addition to the support of blank nodes for grouping properties, blank nodes pose another challenge for graph construction in RDF query languages: Naively, one might generate one result instance for each blank node in the variable assignments. However, in many cases this leads to unnecessary large result graphs.

E.g., consider the assignment set $\{(R \to \mathsf{http://w3.org/}, P \to \mathsf{director}, O \to \mathsf{"Tim\ Berners\text{-}Lee"}), (R \to \mathsf{http://w3.org/}, P \to \mathsf{director}, O \to \mathsf{\_:1})\}$. Then the above SPARQL query constructs a graph containing two statements, one stating that the W3C has director “Tim Berners-Lee” and one stating that the W3C has some (unknown or unspecified) creator. However, the second statement is entailed by the first one and therefore superfluous. A **minimal** result graph would only retain those blank nodes that are not “compatible” and thus entailed by the other resources in the graph.

Conditional construction. When constructing a result graph, the shape of the graph is often closely linked to the variable assignments. This goes, again, beyond mere instantiation of variables at predefined positions. E.g., one might only want to include a subgraph if a certain optional variable is bound. This ability of a query language is referred to as conditional construction. One can essentially distinguish three forms of conditional construction:

1. *Unscoped optional construction* is used, e.g., in SPARQL: A triple containing optional variables is only included if bindings for all optional variables are provided in the current variable assignment. The drawback of this approach is that it does not allow the existence of a binding for an optional variable to have effect beyond triples using that variable. E.g., it is not possible to add the statement that a resource is (of type) translated if a translator exists.
2. *Scoped optional construction* allows this sort of queries by providing an explicit optional construction construct (e.g., optional in Xcerpt construct terms) with a scope. In RDF, this scope is usually a set of triples that are to be included if a binding for the optional variable is present. In contrast to the first case, not all of these triples have to contain the optional variable.
3. *Full conditional construction* finally uses conditional constructs such as if . . . then or case with arbitrary boolean expressions over the query variables. E.g., one might want to add the triple `?P rdf:type my:Teen` for persons

with `?Age` between 12 and 18 and the triple `?P rdf:type my:Adult` for older persons.

Notice, that all three forms can be expressed if the query language allows disjunction to span selection and construction as is the case in most rule-based query languages such as Xcerpt, Algae, or Triple. In SPARQL, however, disjunction is limited to selection (i.e., WHERE clauses) thus making (2) and (3) inexpressible in SPARQL.

Construction of XML Results If one looks at the RDF data access use cases [35] and considers often cited usage for RDF query languages, the need for a bridge between RDF queries and XML processing becomes evident. Some languages address this by integrating RDF and XML querying, e.g., Xcerpt or approaches such as [87]. Such languages become versatile in the sense of [27].

Most RDF query languages, however, do not consider the intertwining of XML and RDF queries. Still, the need for at least a means to deliver XML as result of an RDF query is evident. SPARQL, e.g., defines a static schema for representing answers in XML, cf. [9]. Such a static schema can then serve for further processing by means of XML query languages or other processing tools.

5.3 Procedural Abstraction

This section closes with a brief look at procedural abstraction mechanisms for RDF query languages. Procedural abstraction in form of database views or rules is a common feature of both programming and expressive query languages. For the Semantic Web to succeed, an efficient rule layer to implement large scale reasoning tasks is essential. Separating querying and (rule) reasoning, however, is often infeasible, in particular if the extent of the queried data depends on the reasoning and is not known a priori (as is the case, e.g., in crawling RDF queries).

In addition, rules or views are useful for the query author for all the reasons traditional procedural abstraction has become commonplace in programming languages (separation of concern, reuse, etc.).

Therefore, quite a number of RDF query languages provide some form of rules or views. TRIPLE and Xcerpt, e.g., use deductive rules similar to Logic Programming or Datalog, Algae uses production rules, cf. Section 6 on the evaluation of these different rule paradigms.

Both, TRIPLE and Xcerpt use rules to provide *transparent RDFS-aware selection* as discussed above in Section 5.1, but also allow the user to define their own rules expressing, e.g., application semantics already on the query layer.

A further important use for rules is the integration and mediation of heterogeneous data. The data may differ in format, schema, or just representation, if the schema is flexible as most RDFS schemata. In these cases, rules can ease data integration, e.g., if mappings between the different schemata are provided in some form, cf. [89]. They can also perform data normalization transparent to the query user, i.e., allow the user to query representational variants without considering all these variants in each query anew.

6 Query Evaluation

Methods for RDF query evaluation differ in several aspects:

- RDF data may be stored in memory or on disk.
- Query evaluation may be distributed over a network of collaborating nodes, or it may be local.
- RDF triples may include provenance information. In this case, they are called quadruples (s, p, o, c) of subject, predicate, object and so-called *context information*. Alternatively, the provenance information may be associated with entire subgraphs rather than with triples.
- RDF graphs can be stored as decomposed triples or quadruples in a relational database engine, as documents on a file system, or as entire graphs in an object oriented or semi-structured database. The type and schema of the storage have a high influence on the efficiency of query processing.
- Queries may either consist of single RDF statements with variables substituted for any combination of subject, predicate and object (e.g. `(?X, foaf:knows, ?Y)`), or they may consist of conjunctions of such statements, then referred to as *conjunctive queries*. In the latter case, multiple occurrences of the same variable are evaluated by joins and allow querying graph patterns.

In this article we mainly focus on non-distributed answering of RDF queries on large RDF repositories stored on disk. Both querying graphs with and without provenance information are discussed, and different storage methods are examined. Both single statement queries and conjunctive queries consisting of multiple RDF statements are considered.

6.1 Storage of RDF Data

The first issue highlighted in the field of query evaluation is data storage: a closer look is taken at three alternative approaches to storing RDF data. First, light is shed on the use of the Berkeley database for storing RDF in the Jena framework, second several proposed methods for using relational database engines for RDF storage are reproduced, and third approaches for deploying object oriented and object relational databases for RDF storage are described. Taking into account their widespread use, it is not surprising that the greatest number of suggestions and implementations of RDF storage is based upon relational database engines. In each of the sections, the impact of the choice of the storage method on query evaluation is highlighted.

RDF Storage in Berkeley Databases According to the directory of the Free Software Foundation[18], the Berkeley Database is

> [..] an embedded database system. Its access methods include B+tree, Extended Linear Hashing, fixed and variable-length records, and Persistent Queues. Berkeley DB provides full transactional support, database recovery, online backups, and separate access to locking, logging and shared memory caching subsystems. [..]

The initial database back-end for the Jena RDF framework [47] supports both relational database back-ends and the Berkeley database. The relational database schema for storing RDF statements in Jena1 (the first version of Jena) is very efficient in space, because it does not contain any redundant information. In contrast, each RDF statement is stored three times in the Berkeley database – using all of subject, predicate and object as hash-keys. According to [99] the redundant storage yields a significant enhancement of query performance, and from this experience the authors of Jena decided to not fully normalize the relational database schema for Jena2 (the second version of the Jena RDF Framework). Besides Jena, also the Redland RDF Application Framework [8], rdfDB and RDFStore make use of the Berkeley database.

Storage of RDF at the aid of Relational Database engines The majority of suggestions for permanently storing RDF data concern relational database engines.

RDF storage in Jena1 and Jena2 The most straight-forward approach to storing RDF in a relational DBS is to create a single table with the columns subject, predicate and object, containing all statements of the RDF graph. In order to save space, the relational database schema of Jena1 differs from this simplistic approach in that the schema is normalized to contain each resource and literal only once. Therefore a *resource table* and a *literals table* are introduced, containing a column for a short primary key, and a column for resources and literals, respectively. The subjects, predicates and objects of the statement table refer to these keys.

Although this schema is very efficient in space, retrieving the subject, object and predicate of a statement already requires three joins between the *statement table*, the *resource table* and the *literals table*. Therefore the relational database back-end of Jena2 [99] stores literals and resources directly in the *statements table* unless they supersede a configurable maximum size. As a result, short URLs may be stored multiple times in order to avoid joins, but large URLs are only stored once in order to save space. There are several other optimizations that have been incorporated into Jena2:

- *Multiple tables for different graphs.* RDF applications may wish to store data which is seldom accessed together in different tables, and data which is often

[18] `http://directory.fsf.org/`

queried together in the same tables. “The use of multiple statement tables may improve performance and caching” [99, Section 3.1].

– *Property tables.* In RDF graphs, there are usually sets of statements with the same subject that occur frequently together. An example would be the properties `foaf:name`, `foaf:nick`, `foaf:knows`, etc. of the FOAF vocabulary. So as to provide efficient access to these *common statement patterns*, they are stored in special *property tables.* For each common statement pattern, one property table is provided, and common statement patterns may be automatically detected in RDF Graphs.
– *Reified statements tables.* In Jena1 reified statements are not stored in their reified form (which would require four ordinary statements for one reified statement), but in the statements table with two extra columns – one of them indicating whether the statement is reified, and the other containing the statement identifier. Since also reified statements constitute common access patterns, Jena2 stores reified statements in property tables.

Storage of RDF data in 3store 3store [50] is a `C`-library developed at the University of Southampton with a MySQL database back-end. It is intended for very large RDF databases and is being tested with over 30 million RDF triples holding knowledge about authors, publications and institutions in UK Computer Science research. The database schema employed is very similar to that of Jena1. It consists of a statements table, and a table for resources and literals. As in Jena1, literals and resources are not directly stored in the statements table. Instead a portion of their MD5 hash values are stored as 64-bit foreign keys in the statements table. The use of the hash function for literals and URIs and the storage in extra tables guarantee lower overall space of the database, few string comparisons, and a uniform length of the records in the statements table, “an optimization which benefits the MySQL database engine” [50, Section 4.3]. Although the probability of hash collisions is very low (10^{-10} for $5 \cdot 10^8$ resources), hash collisions are detected and reported at assertion time. [50] does not mention how hash collisions are corrected. Hash collisions among homonymous literals and URIs are averted by splitting the hash space into two equally large parts, one for literals, the other for URIs.

The most recent version of 3store [49] allows the formulation of queries in SPARQL, which supports the concept of named graphs. Therefore, the statements table contains an additional row which indicates the graph that the statement belongs to (triples with such provenance information are often called *quads.* Besides the statements table, and the tables for literals and URIs, 3store also stores the languages and data types of literals in special tables.

RDF Storage in Sesame Sesame is an RDF database with support for Schema inferencing and querying using the SeRQL query language. By introducing an additional *Storage and Inference Layer* (SAIL) between the RDF storage system and the applications accessing the data, Sesame is designed to support a wide variety of different storage possibilities. In [23] an implementation of SAIL in the open source databases PostgreSQL and MySQL is presented.

The PostgreSQL database schema makes use of transitive sub-table relations, which are a special PostgreSQL feature, to model RDFS' property and class subsumption hierarchies. A table holding instances of a class C_1 which is a subclass of class C_2 inherits from the table for C_2 – in other words it is declared as a sub-table of C_2. A query issued on the contents of table C_2 is also evaluated on the entries of table C_1. As Jena1 and 3store, Sesame stores resource URIs and literal values only once to save space. An important difference between Sesame RDF storage and the solutions discussed so far is that statements are not stored in a single statements table consisting of subject, property and object. Instead, an extra table is created for each property and class which is used in the RDF graph. Since this procedure requires the insertion of new tables to the schema when RDF statements are added which use properties or classes which have not appeared in the RDF graph so far, we call these kinds of schemas *dynamic schemas* as opposed to *static schemas* as used in 3store and Jena. An RDF graph with FOAF data would thus include tables `foaf:knows` containing all pairs of person URIs for persons knowing each other, tables `foaf:name`, `foaf:nick` for storing ordinary names and nick names, etc. are created. For each class used in the RDF schema, tables such as `foaf:Person`, `foaf:Document`, etc. Data about the schema is stored in special tables `rdfs:Class`, `rdfs:Property`, `rdfs:domain`, `rdfs:range`, etc. A performance comparison with a *static* PostgreSQL schema has shown, that schemas with a single statement table are faster when inserting or updating data from the RDF graph. Especially the insertion of new `rdfs:subClassOf` statements is expensive, since it requires rebuilding the parts of the subclass-hierarchy modeled by PostgreSQL sub-tables. On the other hand, the authors of [23] expect querying to be faster in the *dynamic* database schema.

The alternative MySQL implementation of the Sesame *Storage and Inference Layer* uses a static database schema. This schema is significantly more complex than the static schemes of 3store and Jena in that it contains tables dedicated to holding the predefined RDF/S properties `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdf:type`, etc. Although not explicitly mentioned in [23], administering this schema information in separate tables enhances the performance of RQL schema queries such as `subClassOf(Artist)`. The fact that RQL is a language that explicitly supports the straightforward formulation of schema queries, and that the other storage engines are coupled with languages with lower support for schema queries may be an explanation for the different database schemas employed.

RDF Storage in RDFSuite RDFSuite is a set of tools for querying, validating and storing RDF data. It natively supports the RQL query language. In this paragraph, its storage system is briefly examined. RDFSuite uses the PostrgreSQL DBS for storing RDF data, and its schema is a dynamic schema resembling the PostgreSQL schema of Sesame. Sub-table relationships are used to implement `subClassOf` and `subPropertyOf`-relationships among classes and properties. Since RQL provides syntactic means specifically geared to querying RDF Schema, such queries must be evaluated quickly. Therefore, the schema information is kept in separate tables such as `subProperty`, `subClass`, `Property`, `Class` and `Type`. In contrast to the schemas described above, Namespaces are stored in a separate `Namespace` table in order to save space. This namespace table is referenced from the other tables. A database is built from an RDF-description using a two phase algorithm: In the first phase, properties and classes occurring within the RDF data are extracted, and from this information the database schema is constructed. In the second phase this schema is populated with the instance data from the RDF file.

Path Based Storage of RDF Data Matono et al. [73] point out that storing RDF graphs as decomposed sets of triples is efficient for evaluating single statement queries, but is inefficient for *path based queries*. Whereas in single statement queries one or two items of *subject, predicate* and *object* are omitted, *path based queries* as defined in [73] are finite sequences of arcs $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ from a source node v_0 to the destination v_k. Answering path based queries of length k at the aid of a single statement table requires $k-1$ joins over the table. So as to improve performance, Matono et al. suggest the following procedure:

- The RDF graph to be queried is separated into five subgraphs named CI, PI, T, DR, G containing the class hierarchy (`rdfs:subClassOf` statements), the hierarchy amongst properties (`rdfs:subPropertyOf`), type information (`rdf:type`), domain and range information of properties and all remaining statements, respectively. Only the paths occurring within G are explicitly saved within an appropriate relational table. For the hierarchical subgraphs CI and PI an interval numbering scheme is applied in order to efficiently answer queries concerning their transitive closures. Since the subgraphs T and DR are flat, it does not make sense to extract paths from them.
- For each resource r in the graph G all paths starting at any root node of G and ending at r are saved. In order to be able to efficiently deal with path based queries that start with a wild card (e.g. "give me all titles of books authored by someone"), path expressions are saved in reverse order. Moreover, only the names of the predicates are reflected within the path expressions, whereas node names are omitted. An example path expressions saved in the database would thus be `'#title<#author`. The relational table containing the path expressions consists of two columns, one holding path identifiers, and the other holding path expressions such as the one given above. In a *resource table*, resources are associated with paths that end at this resource.
- Path queries are evaluated by concatenating their predicate names in reverse order and subsequently comparing the resulting string with the path expressions stored in the path expressions table.

The authors of [73] present a performance comparison with the Jena2 framework which suggests that for path queries of length greater than 3, path based storage of RDF data allows significantly faster query processing. For queries of length 1 and 2, Jena2 performed better. The *resource table* associating resources with path identifiers is significantly larger than the actual number of resources, especially in the case of deep and densely interwoven graphs. A further issue not addressed within [73] are path queries that do not start with wildcard nodes (e.g. "Find all titles of books and their authors"). Since the stored paths only contain predicates and no node identifiers, answering such queries still requires joins over the statements table.

RDF Storage in Object Databases In [20] Bönström et al. propose to directly store RDF graphs modeled in an object oriented programming language in an object oriented database (OODB). They compare the performance of all kinds of queries including schema and hybrid queries expressed in RQL on top of the OODBS Fastobjects with the performance of the same queries on top of the relational MySQL database back-end of Sesame. Due to the similarity of RQL and OQL, RQL queries can be straightforwardly translated to OQL. All resources (URIs for nodes and predicates as well as literals) are represented as objects, and the statements of the RDF graphs are stored in the OODB as "an object/reference structure". The performance comparisons conducted in [20] suggest that directly storing an RDF graph in an OODB system considerably speeds up query evaluation, especially for schema and hybrid queries. Performance comparisons with the PostgreSQL back-end of Sesame and other RDF storage systems mentioned above have not been mentioned in the article.

Index Structures for RDF The approaches considered so far use standard database management systems (OODBS and RDBS) or standard libraries (Berkeley DB) to efficiently store and retrieve RDF data on disk. However, some research has already been carried out on developing index structures specifically aimed at RDF. In [72] Matono et al. propose to use suffix arrays to efficiently find paths in RDF graphs. In [52] index structures for RDF statements with context information (also called RDF quads or RDF triples with provenance information). In this section, both of these approaches are briefly reviewed and discussed.

Indexing RDF and RDF Schema with Suffix Arrays Suffix Arrays [68] are index structures used to search for a pattern P of length p in a larger string M of length m. All suffixes of M are sorted in lexicographical order, and the suffix array is efficiently stored as the string M and a sequence of *indexing points* $p_1, \ldots, p_m$ where $p_i, 1 \leq i \leq m$ is the position of the ith suffix (in lexicographical order) in the original string m. Suffix arrays allow to find all instances of P in M in $O(p \cdot logm)$.

Matono et al. propose to extract all paths from an acyclic RDF graph that start at root nodes (nodes without incoming edges) and end at leaf nodes (nodes without outgoing edges) and to represent them as strings by concatenating their labels (or identifiers for their labels). The alphabet Σ of these strings is thus the set of resource URIs and literal values of the Graph. They define the notion of suffix arrays for directed acyclic graphs as a list of indexing points

$(pa_1, po_1), \ldots, (pa_l, po_l)$ where pa_i denotes the path that the ith suffix (in lexicographical order) appears in, and po_i denotes the position within pa_i. Paths within the queried RDF graph matching a particular path query can be found by performing binary searches on the suffix array.

In order to cope with schema queries efficiently, Matono et al. divide the RDF graph into several subgraphs according to the type of predicates, see [72] for details. Performance evaluations presented in [68] indicate that depending on the type of path queries, the proposed indexing scheme speeds up query execution by a factor in between two and nine.

Index Structures for RDF Quadruples Web applications processing data from several different resources usually are interested in tracing where the information originated from in order to judge its trustworthiness. Furthermore, it is often desirable to perform substring searches on large amounts of Semantic Web data. While RDF storage systems making use of the Berkeley database get by with three hashes for the efficient look-up of triples for two given items of the triple, [52] suggest index structures for efficiently searching for substrings (*keyword index*) within resource and literal values and for looking up quadruples (*quad indexes*) based on any combination of subject, predicate, object and context information.

Since resources and literals are referenced from both the keyword index and from the quad indexes, nodes in the RDF graph are mapped to shorter object identifiers which are stored in the indexes instead. Substring matches are determined by using an inverted index on all words appearing as tokens within the queried RDF graph. The inverted index allows to look up lists of object identifiers of resources that a given word appears in and also provides occurrence counts for the words that can be used to optimize the join order of conjunctive queries.

The quad indexes allow to efficiently look up RDF quadruples matching a given query quadruple in which some of the four entries may be omitted. Query quadruples such as (?:`rdf:type`:?:`http//example.com/stmts.rdf`), which finds all `rdf:type` statements originating from the context `http//example.com/stmts.rdf`, can be categorized into $2^4 = 16$ access patterns, depending on which of the four elements of the quadruple are given. A naive implementation would construct 16 indexes to allow the efficient evaluation of queries falling in any of the 16 categories, but Harth et al. show that by taking advantage of prefix queries in B+-trees, only 6 “combined” indexes suffice for this purpose.

6.2 Schema- and Reasoning-aware RDF Querying

As has been pointed out in Section 4, RDF languages can be distinguished by the fact whether they provide constructs taking advantage of RDF/S and OWL reasoning. While the major part of languages does not provide direct means of finding e.g. all subclasses of a given class, or all instances of a class, others do provide such features (e.g. RQL).[19] But also the languages of the SPARQL family do not reject the RDF/S semantics, but simply maintain that the computation of derived facts should be provided by an underlying graph model (e.g., by pre-materialization or on-the-fly construction performed by the storage layer). Therefore, an overview over several approaches of implementing especially the RDF/S semantics are given in this section.

One step in this direction that has already been discussed in Section 6.1, is the use of dynamic relational database schemes containing tables for each defined `rdfs:Class` holding all the instances of the class. This allows to efficiently retrieve all instances of a given class. Additionally, the use of sub-table relationships within database schemes allows the implementation of the `rdfs9` inference rule as defined in [53]. Other RDF/S inference rules have not been covered so far. There are mainly three approaches that deal with the implementation of the RDF/S semantics:

- *Labeling schemes* can help to implement the RDF/S entailment rules concerning the `rdfs:subClassOf` and `rdfs:subPropertyOf` relationships, and any other relationship that is defined to be transitive.
- *Precomputation of derived facts (forward chaining).* Forward chaining can be used to precompute implied RDF statements, not contained in the original RDF graph that are derived from any of the RDF/S rules or even from user-defined rules. This approach trades memory space for execution time, and is especially useful, if the queried graph and its schema information are stable and if the number of queries issued on the graph is high. Note, however, that this approach requires that RDF triples and therefore the Web sites involved are known beforehand. Indeed, this computation model is not suitable for *crawler queries* where the extent of the data is extended at query time. However, since many RDF query languages including SPARQL and RDQL do not support such queries the computation model is relevant for RDF querying.
- *Backward chaining.* Like forward chaining, backward chaining can be used to implement any kind of rules including all RDF/S entailment rules. Backward chaining is preferred when the underlying graph changes frequently, and when the the number of queries is low. Xcerpt uses backward chaining in combination with simulation unification to evaluate programs. The evaluation of Xcerpt is not treated in this article for the sake of brevity, cf. [28, 90].

Labeling Schemes for RDF/S Reasoning Christophides et al. advocate the use of labeling schemes in conjunction with relational database storage of RDF graphs for "avoiding costly transitive closure computations over voluminous class hierarchies"[32] in Semantic Web data bases such as the Open Directory Portal.

[19] Note that none of the examined languages provides constructs for taking advantage of OWL semantics. However, some research on how to combine query languages with OWL reasoners has already been carried out.

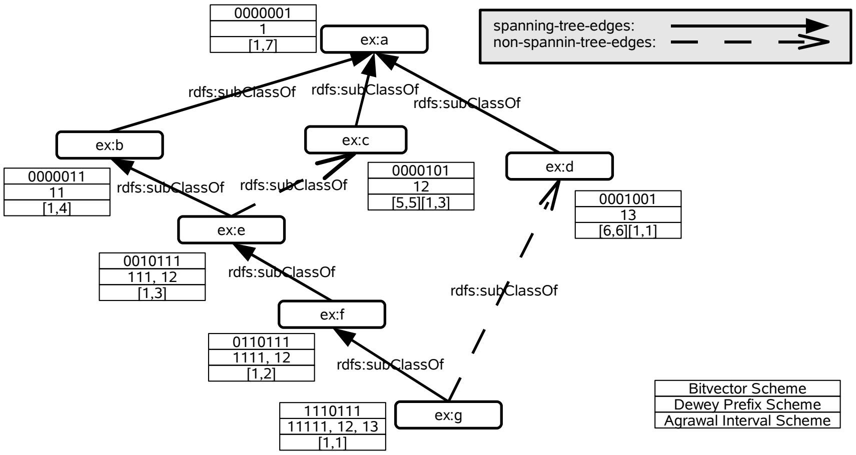
The use of labeling schemes reportedly results in a decrease in query execution time for transitive closure computations of 3-4 orders of magnitude compared to evaluating such queries on a dynamic relational database scheme such as the one described in [55].

In [32] three types of labeling schemes are compared with respect to their suitability for supporting ancestor/descendant (which is a more general form of subclass queries), adjacent/sibling, and nearest common ancestor queries. Some of the results concerning the use of these labeling schemes for both hierarchical subsumption relationships and those structured as directed acyclic graphs (DAGs) are recapitulated here and an example is given in Figure 3.

- *Bitvector schemes* assign bitvectors of length n (n is the number of nodes within a DAG to be represented by the scheme) to the nodes. The ith node in the DAG has a 1 bit at the ith position, and a 1 bit at the position k, if the kth node is one of its ancestors. All other positions within its bitvector are 0. Bitvector schemes allow subsumption checking in constant time (the length of the bitvectors is assumed to be constant), but finding all ancestors, descendants or siblings can only be achieved in $O(n)$. Additionally, the size of the bitvector must be adjusted, when new classes are added to a class hierarchy, making this method inappropriate for class hierarchies in the presence of dynamic updates. As Figure 3 shows, the bitvector scheme can be naturally extended to account for multiple inheritance among RDF classes.
- *Prefix schemes* assign labels to nodes within a class hierarchy (or DAGs in general), such that for each node N and an arbitrary ancestor A the label of A is a prefix of the label of N. Probably the most known representative of prefix schemes is the Dewey Decimal Encoding (DDE). A major advantage of prefix schemes is their support for dynamic updates. New sibling nodes can be added as long as the total number of siblings does not exceed the size of the alphabet chosen (in the figure the alphabet is $\Sigma = \{1, \ldots, 9\}$). The major disadvantage is the inflationary label size for class hierarchies which are not tree-shaped: Each non-spanning-tree edge in Figure 3 causes the node it originates from and all of its descendants to inherit the label of the node the non-spanning-tree edge points at.
- *Interval schemes* assign lower and upper bounds to nodes, such that for a node N and an arbitrary ancestor A, the interval of N is contained within the interval of A, and for two sibling nodes the intervals are disjoint.In the interval based labeling scheme of Agraval et al., the label of a node v is composed of a pair of numbers $(min(v), post(v))$ where $post(v)$ is the post-order number of the node and $min(v)$ is the minimal post-order number of the descendants of v. As shown in Figure 3, the labeling scheme by Agrawal et al. can also be extended to handle DAGs. In contrast to the downward propagation of labels in the prefix schemes, labels are propagated upwards when non-spanning-tree edges are to be reflected (e.g. the node `ex:d` inherits the label of the node `ex:g` because there is a non-spanning-tree edge from

`ex:g` to `ex:d`. The top node `ex:a` does not need to inherit the label `[1,1]` of `ex:g`, since `[1,1]` is already included in the interval `[1,7]` of `ex:a`.

Fig. 3 Labeling schemes for DAG sub-class hierarchies



Note that all of the above labeling schemes cannot be used to represent cycles in the subsumption graph. An alternative labeling scheme for graphs with cycles is the 2-hop labeling [36].

Forward Chaining The most apparent approach to calculating the transitive closure of `rdfs:subPropertyOf` and `rdfs:subClassOf` relationships and other implied RDF statements derivable by inference rules is the following: The body of a rule is instantiated with facts from the knowledge base, such that it becomes true (if possible) and the instantiated head of the rule is added to the knowledge base if it is a new fact. In this way, each of the rules is applied to the knowledge base in turn, until a complete run over the rules does not produce any new derived statements. Once that the application of all rules does not produce any new statements, one can be sure that all implied RDF/S statements have been added to the knowledge base.

Let F be the number of facts, R the number of rules and C the average number of conditions within the head of the rules. Then the maximum number of comparisons between facts and conditions for one loop over the rules is $R*F^C$. The overall complexity additionally depends on the number of loops that need to be performed. Several proposals for improving runtime behavior can be thought of:

– Applying the rules to the entire knowledge base in each round is not necessary. It suffices to consider only those instantiations of the inference rules that make use of a *new* fact – that means a fact that has been added after the last application of the rule. In doing so, the specific semantics of RDF blank nodes should be considered.
– If the body of an inference rule could almost be completely instantiated in one round, the information about the successfully instantiated part gets lost before the rule is reconsidered. By remembering partial instantiations of rule bodies one can treat space for time.
– Especially in the case that rules are complex, the bodies of different rules may share common parts of the condition. In the naive algorithm these subconditions are evaluated once for each rule.

Note, that forward chaining might be difficult to realize if the Web sites involved and thus the RDF facts are not all known before hand as is the case, e.g., with crawler queries.

CWM and Pychinko CWM[20] (an acronym for Closed World Machine) is a Python command line tool for RDF documents that can – amongst other things – convert between different formats (currently the serializations Notation3, RDF/-XML and NTriples are supported) and store triples in a queryable database. The more interesting feature of CWM for this section is its ability to do forward chaining. Given the following rule and data, CWM will infer that `:Frank`, `:Bob`, and `:Sam` are `:Male` (the shorthand `a` represents an `rdf:type` property).

```
{ ?x :son ?y } => { ?y a :Male }.

:Mary :son :Frank, :Bob, :Sam.
```

Since CWM does not employ any optimization techniques for forward chaining, it does not perform very well on large sets of assertions and rules. The authors of Pychinko[21], a CWM clone, improved the performance of CWM by implementing the RETE-algorithm [42].

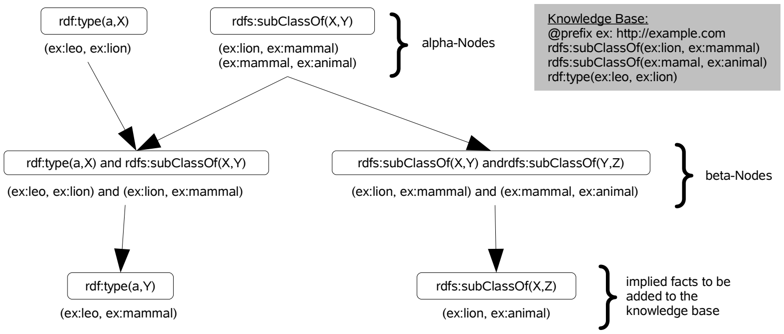
The RETE-Algorithm The RETE-algorithm was conceived by Charles Forgy at Carnegie Mellon University in 1979, and formed the basis for new developments in the ambit of expert systems. Its core idea is to (1) merge (parts of) the antecedents of rules if they are the same, to (2) memorize possibly partial instantiations of antecedents of rules, and to only consider new facts within each loop over the set of rules. The data structure at the core of the RETE algorithm is a network computed from the antecedents of the rules. An example of this data structure for RDF/S entailment rules and some RDF/S statements is given in Figure 4. The network reflects the RDF/S inference rules `rdfs9` and `rdfs11` and contains two kinds of nodes: α-nodes representing simple conditions

[20] `http://infomesh.net/2001/cwm/`

[21] `http://www.mindswap.org/~katz/pychinko/`

and β-nodes representing conjunctions over α-nodes. The α-nodes are populated with matching facts from the knowledge base (an RDF graph), and beta nodes are populated if a conjunction of simple conditions becomes true. The set of initially known facts is given at the top right of Figure 4. Note that although `rdfs9` and `rdfs11` are very simple entailment rules, the principles of the RETE algorithm already allow for some optimizations. Both rules share a common antecedent (the node `rdfs:subClassOf(X,Y)`), and partial instantiations of rules are memorized (e.g. the instantiation `(ex:mammal, ex:animal)`, which will help to derive additional implied RDF statements in the next loop).

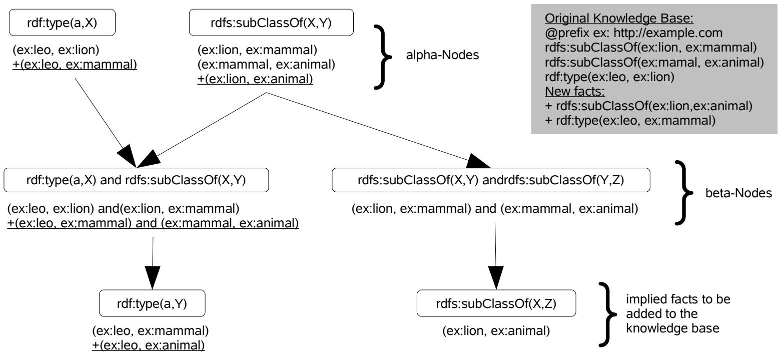
Fig. 4 Memorization of partially instantiated antecedents and combination of rule antecedents in RETE algorithm


rdf:type(a,X)
(ex:leo, ex:lion)
rdfs:subClassOf(X,Y)
(ex:lion, ex:mammal)
(ex:mammal, ex:animal)
alpha-Nodes
Knowledge Base:
@prefix ex: http://example.com
rdfs:subClassOf(ex:lion, ex:mammal)
rdfs:subClassOf(ex:mamal, ex:animal)
rdf:type(ex:leo, ex:lion)
rdf:type(a,X) and rdfs:subClassOf(X,Y)
(ex:leo, ex:lion) and (ex:lion, ex:mammal)
rdfs:subClassOf(X,Y) andrdfs:subClassOf(Y,Z)
(ex:lion, ex:mammal) and (ex:mammal, ex:animal)
beta-Nodes
rdf:type(a,Y)
(ex:leo, ex:mammal)
rdfs:subClassOf(X,Z)
(ex:lion, ex:animal)
implied facts to be added to the knowledge base

As Figure 4 shows, the new facts `rdf:type(ex:leo, ex:mammal)` and `rdfs:subClassOf(ex:lion,ex:animal)` can be inferred. Adding these new facts to the knowledge base as in Figure 5 shows that the amount of comparisons to be performed is low: The derived facts must only be compared with the two α-nodes, and trigger one new instantiation for each α-node and a new instantiation for the left β-node, such that the last implied statement `rdf:type(ex:leo, ex:animal)` can be derived. Note that also the removal of facts (RDF statements) from the knowledge base (the RDF graph) can be efficiently handled by the RETE-algorithm in the same way as the addition of new facts.

Although the optimizations of the RETE algorithm have a greater impact for a large number of rules with complex antecedents, its implementation in Pychinko allegedly yields a five-fold performance increase. Therefore its application to larger and more involved rules for Semantic Web reasoning seems to be promising.

Fig. 5 Addition of new facts to the rete decision tree


rdf:type(a,X)
(ex:leo, ex:lion)
+(ex:leo, ex:mammal)
rdfs:subClassOf(X,Y)
(ex:lion, ex:mammal)
(ex:mammal, ex:animal)
+(ex:lion, ex:animal)
alpha-Nodes
Original Knowledge Base:
@prefix ex: http://example.com
rdfs:subClassOf(ex:lion, ex:mammal)
rdfs:subClassOf(ex:mamal, ex:animal)
rdf:type(ex:leo, ex:lion)
New facts:
+ rdfs:subClassOf(ex:lion,ex:animal)
+ rdf:type(ex:leo, ex:mammal)
rdf:type(a,X) and rdfs:subClassOf(X,Y)
(ex:leo, ex:lion) and(ex:lion, ex:mammal)
+(ex:leo, ex:mammal) and (ex:mammal, ex:animal)
rdfs:subClassOf(X,Y) andrdfs:subClassOf(Y,Z)
(ex:lion, ex:mammal) and (ex:mammal, ex:animal)
beta-Nodes
rdf:type(a,Y)
(ex:leo, ex:mammal)
+(ex:leo, ex:animal)
rdfs:subClassOf(X,Z)
(ex:lion, ex:animal)
implied facts to be
added to the
knowledge base

7 Conclusion

Although this survey only considers a (subjectively chosen) subset of the RDF query languages proposed so far, it makes quite clear that the research community has not yet settled on a dominant paradigm to querying Semantic Web data and that this field of research is changing quite quickly. Language constructs and approaches to querying RDF differ both in their availability (e.g. regular path expressions) and also in their exact semantics (e.g. the optional construct). The widespread use of the query languages within Semantic Web projects, which will most probably take place within the upcoming years, will allow to judge the real-world utility of the presented approaches and constructs and will ultimately establish the most usable amongst them.

This article presents some interesting methods for accelerating RDF query evaluation. With the amount of available Semantic Web data increasing exponentially, evaluation methods, efficient storage and retrieval and index structures specifically aimed at RDF become more important for realizing any of the proposed languages.

Acknowledgments

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`).

Bibliography

[1] RDFQL Database Command Reference. Online only, 2004.

[2] iTQL Commands. Online only, 2004.

[3] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the ACM*, 45(5):798–842, 1998.

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

[5] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In J. Maluszinsky and N. Eisinger, editors, *Reasoning Web Summer School 2005*, pages 35–133. Springer-Verlag, LNCS 3564, 2005.

[6] 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.

[7] D. Beckett. *Turtle - Terse RDF Triple Language*, February 2004.

[8] D. Beckett. The Design and Implementation of the Redland RDF Application Framework. 2001.

[9] D. Beckett and J. Broekstra. *SPARQL Query Results XML Format*. W3C, 2006.

[10] D. Beckett and B. McBride. *RDF/XML Syntax Specification (Revised)*. W3C, 2004. URL `http://www.w3.org/TR/rdf-syntax-grammar/`.

[11] 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.

[12] 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.

[13] 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.

[14] T. Berners-Lee. Notation 3, an RDF language for the Semantic Web. Online only, 2004.

[15] T. Berners-Lee. N3QL—RDF Data Query Language. Online only, 2004.

[16] 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.

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

[18] C. Bizer. TriQL—A Query Language for Named Graphs. Online only, 2004.

[19] O. Bolzer. Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/Master thesis, University of Munich, 2 2005. URL `http://www.pms.ifi.lmu.de/publikationen#DA_Oliver.Bolzer`.

[20] V. Bönström, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *LA-WEB*, pages 27–36. IEEE Computer Society, 2003.

[21] 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/`.

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

[23] J. Broekstra, A. Kampman, and F. Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. International Semantic Web Conference*, 2002.

[24] 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.

[25] 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.

[26] 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. URL `http://rewerse.net/publications/index.html#REWERSE-DEL-2004-I4-D2`.

[27] 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.

[28] F. Bry, A. Schroeder, T. Furche, and B. Linse. Efficient Evaluation of n-ary Queries over Trees and Graphs. Submitted for publication, 2006.

[29] 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.

[30] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. Workshop on Web and Databases*, 2000.

[31] 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.

[32] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On Labeling Schemes for the Semantic Web. In *WWW*, pages 544–555, 2003.

[33] J. Clark and S. DeRose. *XML Path Language (XPath) Version 1.0.* W3C, 1999.

[34] K. Clark. *RDF Data Access Use Cases and Requirements.* W3C, 2004.

[35] K. G. Clark. RDF Data Access Use Cases and Requirements. Working draft, W3C, 10 2004.

[36] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[37] I. F. Cruz, V. Kashyap, S. Decker, and R. Eckstein, editors. *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*, 2003.

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

[39] J. de Bruijn, E. Franconi, and S. Tessaris. Logical Reconstruction of RDF and Ontology Languages. In *Workshop on Principles and Practice of Semantic Web Reasoning*, volume 3703 of *LNCS*. Springer-Verlag, 2005.

[40] 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.

[41] 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.

[42] C. L. Forgy. *On the efficient implementation of production systems.* PhD thesis, 1979.

[43] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In *Proc. International Conference on Very Large Databases*, 1994.

[44] L. M. Garshol. Living with Topic Maps and RDF. Online only, 2003.

[45] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

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

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

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

[49] S. Harris. SPARQL query processing with conventional relational database systems, 2005.

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

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

[52] A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web, 2005.

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

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

[55] 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.

[56] 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.

[57] 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.

[58] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object Oriented and Frame Based Languages. *Journal of ACM*, 42:741–843, 1995.

[59] 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/`.

[60] M. Lacher and S. Decker. On the Integration of Topic Maps and RDF Data. In *Proc. Extreme Markup Languages*, 2001.

[61] M. Lacher and S. Decker. RDF, Topic Maps, and the Semantic Web. *Markup Languages: Theory and Practice*, 3(3):313–331, December 2001.

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

[63] 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/`.

[64] 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.

[65] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. Rul: A declarative update language for rdf. In *Proceedings Int'l. Semantic Web Conf. (ISWC)*, 2005.

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

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

[68] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *SODA*, pages 319–327, 1990.

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

[70] 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.

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

[72] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. An indexing scheme for rdf and rdf schema based on suffix arrays. In Cruz et al. [37], pages 151–168.

[73] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A Path-based Relational RDF Database. 2005.

[74] 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.

[75] 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.

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

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

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

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

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

[81] B. Parsia. Querying the web with sparql. In P. Barahona, F. Bry, E. Franconi, U. Sattler, and N. Henze, editors, *Reasoning Web, Second Int'l. Summer School 2006, Tutorial Lectures*. Springer-Verlag, 2006.

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

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

[84] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. Working draft, W3C, 4 2006.

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

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

[87] 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.

[88] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004. URL `http://www.pms.ifi.lmu.de/publikationen/#PMS-DISS-2004-1`.

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

[90] A. Schroeder. An Algebra and Optimization Techniques for Simulation Unification. Diplomarbeit/Master thesis, Institute for Informatics, University of Munich, 2005. URL `http://www.pms.ifi.lmu.de/publikationen#DA_Andreas.Schroeder`.

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

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

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

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

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

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

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

[98] 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.

[99] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena, 2003.

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