Quo Vadis, Web Queries?

(Invited Tutorial)

Klara Weiand
Institute for Informatics
University of Munich, Germany
http://pms.ifi.lmu.de/~weiand

Tim Furche
Institute for Informatics
University of Munich, Germany
http://furche.net/

François Bry
Institute for Informatics
University of Munich, Germany
http://pms.ifi.lmu.de/~bry

***Abstract*—Various query languages for Web and Semantic Web data, both for practical use and as an area of research in the scientific community, have emerged in recent years. At the same time, the broad adoption of the internet where keyword search is used in many applications, e.g. search engines, has familiarized casual users with using keyword queries to retrieve information on the internet. Unlike this easy-to-use querying, traditional query languages require knowledge of the language itself as well as of the data to be queried. Keyword-based query languages for XML and RDF bridge the gap between the two, aiming at enabling simple querying of semi-structured data, which is relevant e.g. in the context of the emerging Semantic Web. This article presents an overview of traditional query languages for XML and RDF, focused on emerging preeminent exemplars in each field, and contrasts these languages with the field of keyword querying for XML and RDF.**

I. Introduction

Few technologies have been as disruptive to the way we process and manage information than the rapid adaption of the World Wide Web. Getting at information is less and less a question of means, station, race, or location. Rather we have to adapt to new challenges: how to find, among the vast stores of knowledge available, the right information for satisfying our information need.

As part of addressing this challenge, we have seen the emergence of Web queries and query languages that provide us with technological interfaces for accessing information on the Web. The aim is to alleviate some of the burden incurred by the ever increasing volume of information automatically or semi-automatically.

When we talk about Web queries, we subsume two rather diverse areas of research and technology: Web search as Google or Yahoo! provide and database-style queries on Web (mostly XML or RDF) data as provided through languages such as XQuery or SPARQL and incorporated in one form or another in most modern database products.

Web search is about discovering information among the vast amount available on the Web: a search engine sifts through an index of, all or a substantial portion of, Web data and filters out what seems most relevant to the query intent, specified through a very simple query interface (usually just a bag of words). In contrast to traditional information retrieval, Web search exploits for finding and ranking relevant documents not only the content of each individual documents but also their relations expressed as hypertext links. Yet this information must be exploitable without sacrificing scalability to millions or (nowadays) billions of documents. Thus, Web Search engines employ PageRank [158] and similar approaches to harvest structural (or link) ranking as well as non-local search terms (e.g., anchor text used to link to a document or tags used to annotate that document) at indexing time only. This allows the actual evaluation of a search request on each document (and its associated results of the harvesting process) independently and thus allows highly parallel (and thus scalable) evaluation of Web searches.

The downside of Web search, even more than in the case of traditional information retrieval, is that the results to a search request are often rather vaguely related to the search intent and, at best, a ranking can be provided. There is, however, no certainty that each and every returned document is actually related to the search request. Only once they are gauged by a human we can be sure of that. Summarizing, Web search allows us to filter down the huge amount of Web data to what is likely related to our search request. The price for the ability to operate on such a diverse and rapidly changing collection of information is that we can never be sure that the results are precisely what is relevant to our search request.

Database-style Web queries (formulated in languages such as XQuery or SPARQL) are, in many respects, the exact dual of Web search: we peak inside of (a small set of) documents to find precise data items such as the price of a book, the capital of a country, etc. These data items can then be processed *automatically*, e.g., to place an order for a book as soon as its price is below a certain threshold. We can also *deduce* or detect "new" knowledge rather than just discover what knowledge is already there: e.g., the number of books someone authored or that there are people who have published in all top five computer science conferences of the current year. Such queries can not be answered by a Web search engine unless that knowledge is already provided a priori. In contrast to the traditional databases, database-style Web queries operate on Web data formats such as XML and RDF, the presumptive foundation for the Semantic Web. Both differ from, e.g., the relational data model first and foremost by more flexible schemata where repetition and recursion are common. This pushes issues such as the influence of tree queries or tree data on query evaluation or efficient reachability queries in trees and graphs to the front, issues that have been treated only

cursorily for relational data.

The price we pay for the ability to precisely select individual data items of a certain characteristic and to automatically process them is twofold: First, compared to Web search interfaces Web query languages such as XQuery or SPARQL are significantly more complex. Writing correct (let alone efficient) Web queries requires significant training and is comparable to a programming task. Second, most Web query languages such as XQuery or SPARQL scale not better than traditional SQL database technology, and thus are clearly unable to process significant subsets of all Web data. Rather, it is unavoidable to preselect a fairly small collection of documents on which to evaluate the queries.

To summarize, where Web search allows us to operate on (nearly) *all the Web*, (database-style) Web queries operate only on a small fraction of the Web's data. Where Web search is limited to *filtering* relevant documents (for human consumption), Web queries allow the precise selection of data items in Web documents as well as their processing, reorganization, aggregation, or deduction of new data. Where Web search can operate on *all kinds* of Web documents at the same time, Web queries are usually restricted to a more homogeneous collection of documents (e.g., only XHTML documents, only DocBook documents). Where Web search requires a *human in the loop* to ultimately judge the relevance of a search result, Web queries allow automated processing, aggregation, and deduction of data. Where Web search can be used by *untrained users*, Web queries usually require significant training to be employed effectively.

Unfortunately, these two areas of research and technology have been mostly separate in the past. Fortunately, this is starting to change in more than one way:

1) Web search engines are beginning to integrate also "peaking" inside Web documents into search results, e.g., to provide the precise answer to "What is the price of milk?" rather than just to point to a document containing that information. For instance, Google integrates querying of structured data about videos, images, and items from Google Base into the search result listing. Yahoo[1] provides similar features, that can be managed by providers of structured data (online bookstores, online movie databases etc.).

2) There has been considerable research into adding information retrieval functionality and primitives to XQuery and similar XML query languages. This effort has culminated in a (candidate) recommendation [7] by the W3C which proposes selection and ranking (or scoring) operators for XQuery inspired by traditional information retrieval. For an overview of relevant articles and proceedings, see recent tutorials on XQuery and XML retrieval [6], [9].

3) The most significant effort towards combining some of the virtues of Web search, viz. being accessible to untrained users and able to cope with vastly heterogeneous data, with those of database-style Web queries is here categorized under the label **keyword-based Web query languages** for XML and RDF documents. Theses languages operate essentially in the same setting as XQuery or SPARQL but with an interface for untrained users instead of a complex programming language. The interface is often (in *label-keyword query languages*) enhanced to allow, e.g., not only a bag-of-word query but some annotations to each word, most notably a context (e.g., only within the author or title of an article). Results are still excerpts of the queried documents, though the precise extent is often determined automatically rather than by the user. Thus, keyword-based query languages trade some of the precision, that languages like XQuery allow the user in formulation exactly what data to select and how to process it, for an easier interface accessible also to untrained or barely trained users. The yardstick for these languages becomes an easily accessible interface (or query language) that does not sacrifice the essential premise of database-style Web queries, that selection and construction are precise enough to *fully automate* data processing tasks.

In this survey we focus on the last mentioned keyword-based Web query language as the most promising direction for combining the ease of use of Web search engines with the automation and deduction features of database-style Web query languages such as XQuery and SPARQL.

To ground the discussion of keyword-based query languages, we first give a concise summary of what we perceive as the main contributions of research and development on Web query languages in the past decade (Section III). This summary is focused specifically on what sets Web query languages apart from their predecessors for traditional (mostly relational) databases. It comes in two parts, one on XML (Section III), one on RDF (Section IV). For XML, we consider three contributions: *reachability* (as expressed, e.g., in XPaths `descendant` axis) in trees, how the restriction to *tree queries* and tree data enables highly efficient query evaluation, and the effect of *order* as a first class concept of the data model. For RDF we consider again three contributions: *reachability* in graphs, dealing with RDF's multi-valued, *optional* properties, and how existential information (or *blank nodes*) affects querying and construction.

In both discussions we also briefly introduce the pre-eminent exemplars of XML, resp. RDF query languages: XQuery and SPARQL. Where illuminating or necessary for the context we also reference other query languages. However, for more extensive introductions into and an extensive comparison of the mentioned query languages (and many more) we refer to previous surveys of XML and RDF query languages [14], [92].

The main part (Section VI) of this survey is dedicated to keyword-query languages, the first such endeavor the authors are aware of: We start with a brief overview of the principles and motivation of keyword-based query languages as well as their relation to Web search. Then we compare existing keyword-based query languages in three groups: In the most basic case, keyword-based query languages are implemented as any other query language (Sec-

[1] http://developer.yahoo.com/searchmonkey/

tion VI-C). However, since keyword-based query languages can also be considered as more easily accessible interfaces to full, traditional query languages, some are implemented by translation into XQuery or SPARQL (Section VI-D). Finally, some approaches consider keyword queries not as an alternative interface but as an enhancement or extension of an existing query language such as XQuery (Section VI-E).

We conclude this survey with a (1) summary of how keyword-based query languages for XML and RDF aim to bring the ease of use of Web search together with the automation and deduction capabilities of traditional Web queries, (2) a discussion where the existing approaches succeed in this aim and what, in our opinion, are the most glaring open issues, and (3) where, beyond keyword-based query languages, we see the need, the challenges, and the opportunities for combining the ease of use of Web search with the virtues of Web queries.

II. Data on the Semantic Web: XML and RDF

A. Extensible Markup Language (XML)

XML [34] is, by now, *the* foremost data representation format for the Web and for semi-structured data in general. It has been adopted in a stupendous number of application domains, ranging from document markup (XHTML, Docbook [188]) over video annotation (MPEG 7 [141]) and music libraries (iTunes[2]) to preference files (Apple's property lists [11]), build scripts (Apache Ant[3]), and XSLT [122] stylesheets. XML is also frequently adopted for serialization of (semantically) richer data representation formats such as RDF or TopicMaps.

XML is a generic markup language for describing the structure of data. Unlike in HTML (HyperText Markup Language), the predominant markup language on the web, neither the tag set nor the semantics of XML are fixed. XML can thus be used to derive markup languages by specifying tags and structural relationships.

The following presentation of the information in XML documents is oriented along the XML Infoset [68] which describes the information content of an XML document. The XQuery data model [85] is, for the most parts, closely aligned with this view of XML documents.

Following the XPath and XQuery data model, we provide a *tree shaped* view of XML data. This deviates from the Infoset where valid ID/IDREF links are resolved and thus the data model is graph, rather than tree shaped. This view is adopted in some XML query languages such as Xcerpt [47] and Lorel [3], but most query languages follow XPath and XQuery and consider XML tree shaped.

1) XML in 500 Words: The core provision of XML is a syntax for representing hierarchical data. Data items are called elements in XML and enclosed in start and end *tags*, both carrying the same tag names or *labels*. `<author>...</author>` is an example of such an element. In the place of '...', we can write other elements or character data as *children* of that element. The following listing shows a small XML fragment that illustrates elements and element nesting:

```
<bib xmlns:dc="http://purl.org/dc/elements/1.1/">
 <article journal="Computer Journal" id="12">
   <dc:title>...Semantic Web...</dc:title>
   <year>2005</year>
   <authors>
    <author>
     <first>John</first> <last>Doe</last> </author>
    <author>
     <first>Mary</first> <last>Smith</last> </author>
  </authors>
 </article>
 <article journal="Web Journal">
   <dc:title>...Web...</dc:title>
   <year>2003</year>
   <authors>
    <author>
     <first>Peter</first> <last>Jones</last> </author>
    <author>
     <first>Sue</first> <last>Robinson</last> </author>
  </authors>
 </article>
</bib>
```

In addition, we can observe *attributes* (name, value pairs associated with start tags) that are essentially like elements but may only contain character data, no other nested attributes or elements. Also, by definition, *element order* is significant, attribute order is not. For instance

```
<author><last>Doe</last><first>John</first></author>
```

represents different information than the `author` element in lines 6–9, but

```
<article id="12" journal="Computer
     Journal">...</article>
```

represents the same element information item as lines 2–15.

Figure 1 gives a graphical representation of the XML document that is referenced in preceding illustrations. When represented as a graph, an XML document without links is a labeled tree where each node in the tree corresponds to an element and its type. Edges connect nodes and their children, that is, elements and the elements nested in them, elements and their content and elements and their attributes. Since the visual distinction between the parent-child relationship can be made without edge labels and since attributes are not addressed or receive no special treatment in the research presented in this text, edges will not be labeled in the following figures.

Elements, attributes, and character data are XML's most common information types. In addition, XML documents may also contain *comments*, *processing instructions* (name-value pair with specific semantics that can be placed anywhere an element can be placed), *document level information* (such as the XML or the document type declarations), *entities*, and *notations*, which are essentially just other kinds of information containers.

[2] http://www.apple.com/itunes/
[3] http://ant.apache.org/

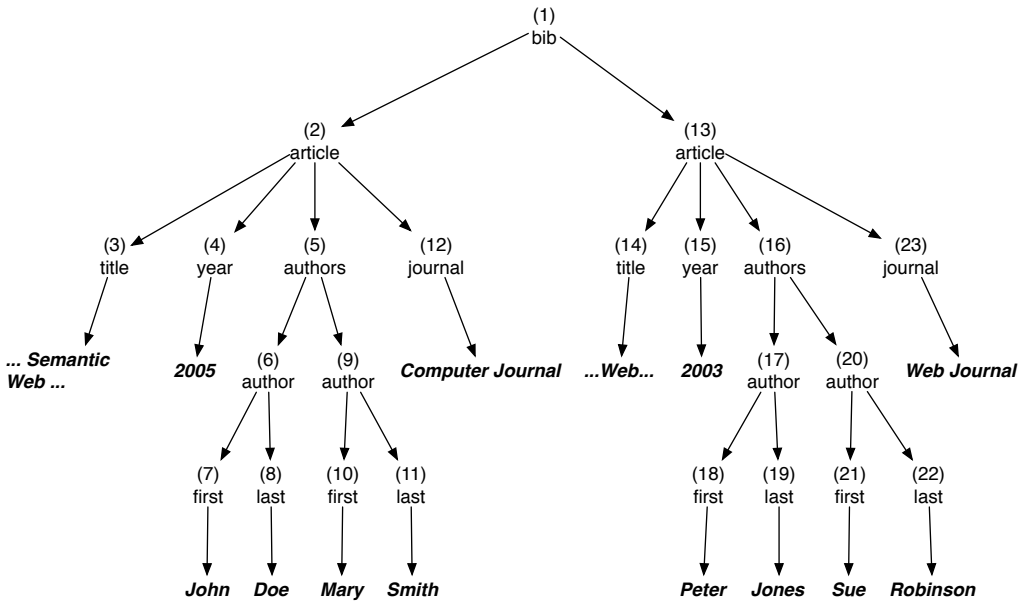

Fig. 1. Visual representation of sample XML document

On top of these information types, two additional facilities relevant to the information content of XML documents are introduced by subsequent specifications: Namespaces [33] and Base URIs [140]. Namespaces allow the partitioning of element labels used in a document into different namespaces, identified by a URI. Thus, an element is no longer labeled with a single label but with a triple consisting of the *local name*, the *namespace prefix*, and the *namespace URI*. E.g., for the `dc:title` element in line 3, the local name is `title`, the namespace prefix is `dc`, and the namespace URI (called "name" in [68]) is `http://purl.org/dc/elements/1.1/`. The latter can be derived by looking for a *namespace declaration* for the prefix `dc`. Such a declaration is shown in line 1: `xmlns:dc="http://...` It associates the prefix `dc` with the given URI in the scope of the current element, i.e., for that element and all elements contained within unless there is another nested declaration for `dc`, in which case that declaration takes precedence. Thus, we can associate with each element a set of *in-scope namespaces*, i.e., of pairs namespace prefix and URI, that are valid in the scope of that element. Base URIs [140] are used to resolve relative URIs in an XML document. They are associated with elements using `xml:base="http://...` and, as namespaces, are inherited to contained elements unless a nested `xml:base` declaration takes precedence.

The above features of XML are covered by most query languages. Additionally some languages (most notably XQuery) also provide access to type information associated via DTD or XML Schema [82]. These features are mentioned below where appropriate but not discussed in detail here.

B. Resource Description Framework (RDF)

As the second preeminent data format on the Semantic Web, the *Resource Description Format (RDF)* [109], [125], [139] is emerging. RDF is, though much less common than XML, a widespread choice for interchanging (meta-) data together with descriptions of the schema and, in contrast to XML, a basic description of its semantics of that data.

Not to distract from the salient points of the discussion, we omit typed literals (and named graphs) from the following discussion.

1) RDF in 500 Words: RDF graphs contain simple statements about *resources* (which, in other contexts, are be called "entities", "objects", etc., i.e., elements of the domain that may partake in relations). Statements are triples consisting of subject, predicate, and object, all of which are resources. If we want to refer to a specific resource, we use (supposedly globally unique) URIs, if we want to refer to a resource for which we know that it exists and maybe some of its properties, we use *blank nodes* which play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. Finally, for convenience, we can directly use *literal values* as objects.

RDF may be serialized in many formats (for a recent survey see [30]), such as RDF/XML [18], an XML dialect for representing RDF, or Turtle [13] which is also used in SPARQL. The following Turtle data represents roughly the same data as the XML document discussed in the previous section:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
```

```
@prefix ex: <http://example.org/libraries/#> .
ex:smith2005 a bib:Article ; dc:title "...Semantic
    Web..." ;
     dc:year "2005" ;
     ex:isPartOf [ a bib:Journal ;
        bib:number "11"; bib:name "Computer Journal" ] ;
     bib:author [ a rdf:Bag ;
        rdf:_1 [ a bib:Person ;
           bib:last "Smith" ; bib:first "Mary" ] ;
        rdf:_2 [ a bib:Person ;
           bib:first "John" ; bib:last "Doe" ] ] .
```

Following the definition of namespace prefixes used in the remainder of the Turtle document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semi-colon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 1 reads as ex:smith2005 is a (has rdf:type) bib:Article and has dc:title "...Semantic Web...". Lines 3–4 show a blank node: the article is part of some entity which we can not (or don't care to) identify by a unique URI but for which we give some properties: it is a bib:Journal, has bib:number "11", and bib:name "Computer Journal".

Figure 2 shows a visual representation of the above RDF data, where we distinguish literals (in square boxes) and classes, i.e., resources that can be used for classifying other resources, and thus can be the object of an `rdf:type` statement (in square boxes with rounded edges) from all other resources (in plain ellipses).

2) Semantics: What sets RDF apart from XML and justifies its role as *the* data format for the *Semantic* Web is that RDF data comes with attached meaning, that allows us to infer additional knowledge beyond what is stated explicitly. Query languages are usually expected to behave consistent w.r.t. some form of RDF entailment (e.g., simple, full, or RDFS entailment), i.e., graphs equivalent under the respective entailment yield the same answers. Simply stated, rather than just consulting the actual RDF data for answering a query, we might also need to consider additional, inferred triples depending on the form of entailment chosen. E.g., when querying for resources of type bib:Publication we might also want to return bib:Articles if we have the additional information that bib:Article is a sub-class of bib:Publication. SPARQL, e.g., is designed to be agnostic of the particular entailment used: it can be used to query RDF data under any of the above mentioned entailment forms.

RDF Interpretations are used to provide meaning to an RDF graph. URIs in subject or object position are interpreted as arbitrary objects, such as people, trains or web pages. An URI in predicate position is interpreted as a set of pairs of objects such as train connections, coauthor relationships or links between webpages. The set of resources that RDF graphs make statements about is called the *domain* of the RDF graph.

Finally blank nodes are used to express existential knowledge or to group information in RDF graphs. Each blank node is interpreted as a domain element but its interpretation is not fixed: An interpretation is a *model* of an RDF graph iff there is an interpretation for the blank nodes such that for every triple, the interpretation of the subject and object is an element of the interpretation of the predicate. An RDF graph g is said to *entail* an RDF graph h if every model of g is also a model of h.

As the definition of an interpretation resembles the definition used in logic, it is possible to view an RDF graph as a formula. This formula has an atom for every triple. URIs and literals are represented by constants, while blank nodes are represented by existential variables. Formally, the notions of RDF vocabulary, RDF graph, (simple) RDF interpretation, and RDF entailment are defined in [109] and recalled briefly in the following:

a) RDF Graph [109]: An *RDF vocabulary* V consists of two disjoint sets called *URIs* U and *literals* L. The *blank nodes* B is a set disjoint from U and L. An *RDF graph* is a set of RDF triples where an *RDF triple* is an element of $(\mathsf{U}\cup\mathsf{B})\times\mathsf{U}\times(\mathsf{U}\cup\mathsf{L}\cup\mathsf{B})$. If $t=(\mathsf{s},\mathsf{p},\mathsf{o})$ is an RDF triple then s is the *subject*, p is the *predicate*, and o is the *object* of t.

The set L of literals consists of three subsets, *plain literals*, *typed literals* and *literals with language tags*. In this work we consider only plain literals (and thus drop IL, the interpretation function for typed literals, see Section 1.3 in [109], in the following definitions).

b) RDF Interpretation [109]: An *interpretation* I of an RDF vocabulary $\mathsf{V}=(\mathsf{U},\mathsf{L})$ is a tuple $(\mathsf{IR},\mathsf{LV},\mathsf{IP},\mathsf{IEXT},\mathsf{IS})$ where IR is a non-empty set of *resources* such that $\mathsf{L}\subseteq\mathsf{LV}\subseteq\mathsf{IR}$, IP is a set of *properties* and $\mathsf{IEXT}:\mathsf{IP}\to 2^{\mathsf{IR}\times\mathsf{IR}}$, and $\mathsf{IS}:\mathsf{U}\to\mathsf{IR}\cup\mathsf{IP}$ are mappings.

Note that as IR and IP are not necessarily disjoint a same URI can be used both as a resource and a property. RDF interpretations are used to assign a truth value to an RDF graph.

RDF assigns a special meaning to a predefined vocabulary, called RDFS vocabulary. For example it is required that $\mathsf{IEXT}(\mathsf{IP}(\mathsf{rdfs:subPropertyOf}))$ is transitive and reflexive. The formulation of theses constraints on RDF interpretation makes use of a notion of a *class*. We have omitted this notion in the definition above for simplicity. The logical core of RDFS has been identified in [150], denoted as ρdf. An RDF interpretation I is a ρdf *interpretation* if I satisfied the constraints specified in Definition 3 in [150].

c) Interpretation of an RDF Graph [109]: Let I be the RDF (ρdf) interpretation $(\mathsf{IR},\mathsf{LV},\mathsf{IP},\mathsf{IEXT},\mathsf{IS})$ and $\mathsf{A}:\mathsf{B}\to\mathsf{IR}$ a mapping. Then $[\mathsf{I}+\mathsf{A}](\mathsf{e})=\mathsf{a}$ if e is the literal a, $[\mathsf{I}+\mathsf{A}](\mathsf{e})=\mathsf{IS}(\mathsf{e})$ if e is a URI, $[\mathsf{I}+\mathsf{A}](\mathsf{e})=\mathsf{A}(\mathsf{e})$ if e is a blank node, and $[\mathsf{I}+\mathsf{A}](\mathsf{e})=\text{true}$ if $\mathsf{e}=(\mathsf{s},\mathsf{p},\mathsf{o})$ is an RDF triple over V, $\mathsf{I}(\mathsf{p})\in\mathsf{IP}$ and $(\mathsf{I}(\mathsf{s}),\mathsf{I}(\mathsf{o}))\in\mathsf{IEXT}(\mathsf{I}(\mathsf{p}))$. Finally $\mathsf{I}(\mathsf{g})=\text{true}$ if there is a mapping $\mathsf{A}:\mathsf{B}\to\mathsf{IR}$ such that $[\mathsf{I}+\mathsf{A}](t)=\text{true}$ for all RDF triples $\mathsf{t}\in\mathsf{g}$.

The semantics of RDF is completed by the notion of entailment: An RDF graph g *RDF-entails* (ρdf*-entails*) an RDF graph h if for all RDF (ρdf) interpretations I, $\mathsf{I}(\mathsf{h})=\text{true}$ if $\mathsf{I}(\mathsf{g})=\text{true}$ [109].

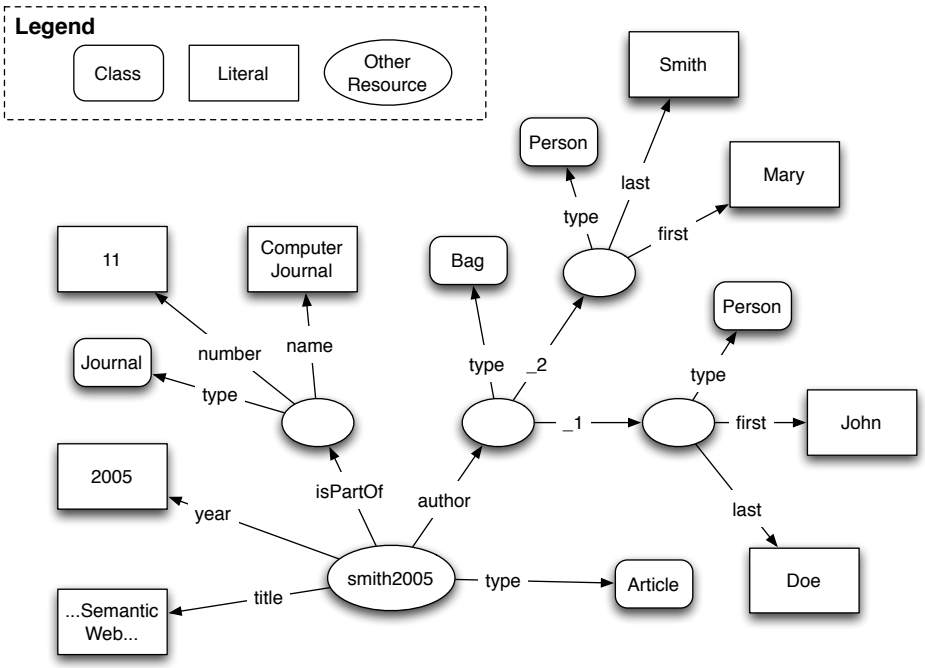

Fig. 2. Visual representation of sample RDF graph

III. Queries as Programs I: XML Query Languages

As discussed in Section II-A, XML is set apart from both from the relational and previous semi-structured data models (as in [3]) by a focus on *ordered tree data*. Both are direct consequences of XML's heritage as a simplified variant of SGML, primarily used for document markup. Documents in formats such DocBook [188] or (X)HTML exhibit an intrinsic hierarchical organisation of the data and are strictly ordered, just like in printed form. It is clearly not acceptable to reorder paragraphs even within the same section, or sections within the same chapter. Though previous (relational or semi-structured) data models allow the modeling of tree data (and sometimes even ordered tree data), XML is the first data format that limits itself to tree data while placing a premium on the maintenance of sibling and document order.

These novelties are reflected well in the contributions of XML query languages over previous approaches and will guide the following discussion. First, we illustrate how XML's focus on tree data pushes the issue of reachability (or descendant and ancestor) queries to the center stage (Section III-D) and how different XML query languages address this issue. Second, we summarize the effect of order as a first class citizen in XML on XML query languages in Section III-F. Finally, we briefly recall how the limitation to tree data and consequently tree queries has yielded a number of novel evaluation strategies tailored to this setting that significantly outperform traditional, less focused approaches.

We start off the discussion of XML query languages with a closer look at two of the more prominent exemplars: XPath and XQuery. We also briefly glance at XML-QL, an early alternative to XQuery that illustrates the strength of patterns rather than paths for multi-variable queries. These introductions are focused on the essentials of these languages necessary for the remainder of this article. For a more in-depth comparison of (more than two dozen) XML query languages see [14].

d) XML trees as relational structures: Following [21], we formalize an XML tree as a relational structure (for defining the semantics of XPath and XQuery): An XML tree is considered a relational structure T over the schema $((\mathsf{Lab}^{\lambda})_{\lambda\in\Sigma}, R_{\mathsf{child}}, R_{\mathsf{nest\text{-}sibling}}, relRoot)$. The nodes of this tree are labeled using the symbols from σ which are queried using $\mathscr{L}^{\lambda}$ (note, that λ is a single label not a label set). The parent-child relations are represented by R_{child}. The order between siblings is represented by $R_{\mathsf{nest\text{-}sibling}}$. The root node of the tree is identified by root. There are some additional derived relations, viz. $R_{\mathsf{descendant}}$, the transitive, $R_{\mathsf{descendant\text{-}or\text{-}self}}$ the transitive reflexive closure of R_{child}, $R_{\mathsf{following\text{-}sibling}}$, the transitive closure of $R_{\mathsf{next\text{-}sibling}}$, R_{self} relating each node to itself, and $R_{\mathsf{following}}$ the composition of $R^{-1}_{\mathsf{descendant\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label. XQuery also considers two more forms of equality: one based on node identity, $=_{\mathsf{nodes}}$ which relates each node to itself, and deep equality $=_{\mathsf{deep}}$ which holds for two nodes if there exists an isomorphism between their respective sub-trees.

For example, the XML document (with id's as subscripts)

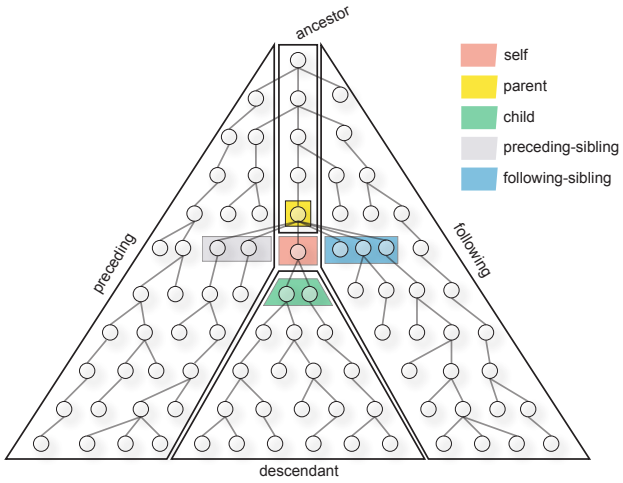

Fig. 3. XPath axis (from [94])

```
<a>1 <b/>2 <c>3<c/>4</c> </a>
```

is represented as $T = (\mathsf{Lab}^a = \{1\}, \mathsf{Lab}^b = \{2\}, \mathsf{Lab}^c = \{3,4\}, R_{\mathsf{child}} = \{(1,2),(1,3),(3,4)\}, R_{\mathsf{nest\text{-}sibling}} = \{(2,3)\}, \mathsf{root} = \{1\})$ over the label alphabet $\{a,b,c\}$. All other relations can be derived from this definition.

In the following, we also allow unions of such structures, i.e., XML "forests".

A. XPath

Unary selection of XML elements is, by now, almost always done using XPath or some variant of XPath (such as XPointer). XPath provides an elegant and compact way of describing "paths" in an XML document viewed as an ordered tree. Paths are made up of "steps" each specifying a direction, called *axis*, in which to navigate through the document, e.g., child, following, or ancestor, cf. Figure 3 for the full set of axes. Together with the axis, a step contains a restriction on the type or label of the data items to be selected, called *node test*. Node tests may be labels of element or attribute nodes, node kind wildcards such as * (any node with some label), element(), node(), text(), or comment(). Any step may be adorned by one or more *qualifiers* each expressing additional restrictions on the selected nodes and denoted with square brackets. Compared to other query languages such as XQuery, SQL, or SPARQL, the most distinctive feature of XPath is the lack of explicit variables. This makes it impossible to express n-ary queries and limits XPath, for the most part, to two-variable logic, see [29], [144] for details.

e) XPath examples: For instance, the XPath expression `/descendant::article/child::author` consists of two steps, the first selecting `article` elements that are descendants of the root ("of the root" is indicated by the leading slash), the second selecting `author` children of such `article` elements. More interesting queries can be expressed by exploiting XPath's qualifiers, e.g., the following XPath expression that selects all authors that are also PC members of a conference (more precisely that have node children with the same label):

```
/child::conference/descendant::article/child::author[. =
    /child::conference/child::member]
```

In addition to the strict axis plus node test notation, XPath uses also an abbreviated syntax where child axis may be omitted, descendant is (roughly) abbreviated by `//`, the current node is referenced by `.` etc. In the following, we only use the full syntax. We also limit ourselves to the core feature of XPath as discussed here and thus present a view of XPath similar to Navigational XPath of [99] and [21]. Due to [155], we also limit ourselves to forward axes such as child and following, rewriting expressions with reverse axes such as parent, ancestor, or preceding where necessary.

f) Syntax of navigational XPath: The syntax of navigational XPath is defined as follows (again following [99] and [21]):

⟨*path*⟩	::=	⟨*step*⟩ | ⟨*step*⟩ '/' ⟨*path*⟩ | ⟨*path*⟩ '∪'⟨*path*⟩
		| '/' ⟨*path*⟩
⟨*step*⟩	::=	⟨*axis*⟩ '::' ⟨*node-test*⟩
		| ⟨*step*⟩'['⟨*qualifier*⟩']'
⟨*axis*⟩	::=	'child' | 'descendant'
		| 'descendant-or-self' | 'next-sibling'
		| 'following-sibling' | 'following'
⟨*node-test*⟩	::=	⟨*label*⟩ | 'node()'
⟨*qualifier*⟩	::=	⟨*path*⟩ | ⟨*path*⟩ '∧'⟨*path*⟩ | ⟨*path*⟩ '∨'⟨*path*⟩
		| '¬'⟨*path*⟩
		| 'lab()' '=' 'λ'
		| ⟨*path*⟩ '=' ⟨*path*⟩

g) Semantics of navigational XPath: The semantics of a navigational XPath expression over a relational structure T representing an XML tree (as defined above) is defined in Table I by means of $[\![\,]\!]_{\mathsf{Nodes}}(n)$ where n is a node, called *context node*. $[\![\,]\!]_{\mathsf{Nodes}}(n)$ associates each XPath expression and context node with a set of nodes that constitutes the semantics of that expression if evaluated with the given context node. It uses $[\![\,]\!]_{\mathsf{Bool}}(n)$ for the semantics of qualifiers under a context node n.

Most parts of the semantics are fairly straightforward: Axes are mapped to their respective relational representations (line 1), paths correspond conjunctions (line 4) where the current (or *context*) variable changes with each step. Qualifiers (treated by $[\![\,]\!]_{\mathsf{Bool}}$) have an inherent existential semantics: as line 6 shows they are true if the contained expression evaluates to a non-empty node-set, i.e., if there is at least one node for which that expression matches.

For more details on the semantics as well as differences to full XPath see [21].

XPath has also been deeply investigated in research. *Formal semantics* for (more or less complete) fragments for XPath have been proposed in [96], [155], [187]. Sur-

$$
\begin{aligned}
[\![axis]\!]_{\mathsf{Nodes}}(n) &= \{(n' : R_{axis}(n, n')\} \\
[\![\lambda]\!]_{\mathsf{Nodes}}(n) &= \{(n' : \mathsf{Lab}^{\lambda}(n')\} \\
[\![\texttt{node()}]\!]_{\mathsf{Nodes}}(n) &= \mathsf{Nodes}(T) \\
[\![axis::nt[qual]]\!]_{\mathsf{Nodes}}(n) &= \{n' : n' \in [\![axis]\!]_{\mathsf{Nodes}} \wedge n' \in [\![nt]\!]_{\mathsf{Nodes}} \wedge \\
&\quad\ [\![qual]\!]_{\mathsf{Bool}}(n')\} \\
[\![step/path]\!]_{\mathsf{Nodes}}(n) &= \{n'' : n' \in [\![step]\!]_{\mathsf{Nodes}}(n) \wedge \\
&\quad\ n'' \in [\![path]\!]_{\mathsf{Nodes}}(n')\} \\
[\![path_1 \cup path_2]\!]_{\mathsf{Nodes}}(n) &= [\![path_1]\!]_{\mathsf{Nodes}}(n) \cup [\![path_2]\!]_{\mathsf{Nodes}}(n)
\end{aligned}
$$

$$
\begin{aligned}
[\![path]\!]_{\mathsf{Bool}}(n) &= [\![path]\!]_{\mathsf{Nodes}}(n) \neq \emptyset \\
[\![path_1 \wedge path_2]\!]_{\mathsf{Bool}}(n) &= [\![path_1]\!]_{\mathsf{Bool}}(n) \wedge [\![path_2]\!]_{\mathsf{Bool}}(n) \\
[\![path_1 \vee path_2]\!]_{\mathsf{Bool}}(n) &= [\![path_1]\!]_{\mathsf{Bool}}(n) \vee [\![path_2]\!]_{\mathsf{Bool}}(n) \\
[\![\neg path]\!]_{\mathsf{Bool}}(n) &= \neg [\![path]\!]_{\mathsf{Bool}}(n) \\
[\![\texttt{lab()} = \lambda]\!]_{\mathsf{Bool}}(n) &= \mathsf{Lab}^{\lambda}(n) \\
[\![path_1 = path_2]\!]_{\mathsf{Bool}}(n) &= \exists n', n'' : n' \in [\![path_1]\!]_{\mathsf{Nodes}}(n) \wedge \\
&\quad\ n'' \in [\![path_2]\!]_{\mathsf{Nodes}}(n) \wedge \cong (n', n'')
\end{aligned}
$$

TABLE I
SEMANTICS FOR NAVIGATIONAL XPATH (FOLLOWING [21])

prisingly, most popular implementations of XPath embedded within XSLT processors exhibit exponential behavior, even for fairly small data and large queries. However, the *combined complexity* of XPath query evaluation has been shown to be P-complete [97], [98]. Various sub-languages of XPath (e.g., forward XPath [155], Core or Navigational XPath [97], [19]) and extensions (e.g., CXPath [142]) have been investigated, mostly with regard to expressiveness and complexity for query evaluation. Also, satisfiability of positive XPath expressions is known to be in NP and, even for expressions without boolean operators, NP-hard [110]. Containment of XPath queries (with or without additional constraints, e.g., by means of a document schema) has been investigated as well, cf., e.g., [77], [148], [179], [195]. For a recent summary of fundamental results on XPath complexity, containment, etc. see [21]. Several methods providing efficient implementations of XPath relying on standard relational database systems have been published, cf., e.g., [101], [104], [156].

Recently, the W3C has, as part of its activity on specifying the XML query language XQuery, developing a revision of XPath: XPath 2.0 [26]. See [121] for an introduction. The most striking additions in XPath 2.0 are: (1) a facility for defining variables (using `for` expressions), (2) sequences instead of sets as answers, (3) the move from the value typed XPath 1.0 to extensive support for XML schema types in a strongly typed language, (4) a considerably expanded library of functions and operators [138], and (5) a complete formal semantics [79].

B. XQuery

Though not nearly as common as XPath, XQuery has nevertheless achieved the status of predominant XML query language, at least as far as database products and research are concerned (in total, XSLT [61] is probably still more widely supported and used). XQuery is essentially an extension of XPath (though some of its axis are only optional in XQuery), but most of XPath becomes syntactic sugar in XQuery. This is particularly true for XPath qualifiers which can be reduced to **`where`** or **`if`** clauses in XQuery. Indeed, the XQuery standard is accompanied [79] by a normalization of XQuery to a core dialect of the language.

h) XQuery Principles: At its core, XQuery is an extension of XPath 2.0 adding features needed to capture all the use cases in [49], i.e., to become a "full query language" and not only a language for (mostly tree-shaped) node selection. The most notable of these features are:

1) *Sequences.* Where in XPath 1.0 the results of path expressions are node sets, XQuery and XPath 2.0 use sequences. Sequences can be constructed or result from the evaluation of an XQuery expression. In contrast to XPath 1.0, sequences cannot only be composed of nodes but also from atomic values, e.g., `(1, 2, 3)` is a proper XQuery sequence.
2) *Strong typing.* Like XPath 2.0, XQuery is a strongly typed language. In particular, most of the (simple and complex) data types of XML Schema are supported. The details of the type system are described in [79]. Furthermore, many XQuery implementations provide (although it is an optional feature) static type checking.
3) *Construction, Grouping, and Ordering.* Where XPath is limited to selecting parts of the input data, XQuery provides ample support for constructing new data. Constructors for all node types as well as the simple data types from XML Schema are provided. New elements can be created either by so-called direct element constructors (that look just like XML elements) or by what is referred to as computed element constructors, e.g. allowing the name of a newly constructed element to be the result of a part of the query. For examples on these constructors, see the implementations for Query 1 and 3 below.
4) *Variables.* Like XPath 2.0, XQuery has variables defined in so-called FLWOR expressions. A FLWOR expression usually consists in one or more `for`, an optional `where` clause, an optional `order by`, and a `return` clause. The `for` clause iterates over the items in the sequence returned by the path expression in its `in` part: `for $book in //book` iterates over all books selected by the path expression `//book`. The `where` clause specifies conditions on the selected data items, the `order by` clause allows the items to be processed in a certain order, and the `return` clause specifies the result of the entire FLWOR expression (often using constructors as shown above). Additionally, FLWOR expressions may contain, after the `for` clauses, `let` clauses that also bind variables but without iterating over the individual data items in the sequence bound to the variable. FLWOR expressions resemble very much XSLT's explicit iteration, selection, and assignment constructs

described above.

5) *User-defined functions.* XQuery allows the user to define new functions specified in XQuery (cf. implementation of Query 3 below). Functions may be recursive.
6) *Universal and existential quantification.* Both XPath 2.0 and XQuery 1.0 provide `some` and `all` for expressing existentially or universally quantified conditions (see implementation of Query 9 below).
7) *Schema validation.* XQuery implementations may (optionally) provide support for schema validation, both of input and of constructed data, using the `validate` expression.
8) *Full host language.* XQuery completes XPath with capabilities to set up the context of path expressions, e.g., declaring namespace prefixes and default namespace, importing function libraries and modules (optional), and (again optionally) providing flexible means for serialization that are in fact shared with XSLT 2.0 (cf. [123]).
9) *Unordered sequences.* As a means for assisting query optimization, XQuery provides the `unordered` keyword, indicating that the order of elements in sequences that are constructed or returned as result of XQuery expressions is not relevant. E.g., `unordered{for $book in //book return $book/name}` indicates that the nodes selected by `//book` may be processed in any order in the `for` clause and the order of the resulting name nodes also can be arbitrary (implementation dependent). Note that inside unordered query parts, the result of any expressions querying the order of elements in sequences such as `fn:position`, `fn:last` is non-deterministic.

In at least one respect, XQuery is more restrictive than XPath: not all of XPath's axes are mandatory, `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, and `preceding-sibling` do not have to be supported by an XQuery implementation. This is, however, no restriction to XQuery's expressiveness, as expressions using reverse axes (such as `ancestor`) can be rewritten, cf. [155], and the "horizontal axes", e.g., `following` and `following-sibling`, can be replaced by FLWOR expressions using the « and » operators that compare two nodes with respect to their position in a sequence.

Comprehensive but easy to follow introductions to XQuery are given in, e.g., [38], [120].

i) Composition-Free XQuery in 1000 Words: In the following, we focus on a fragment of XQuery, called non-compositional XQuery [20], [126], that has a well-defined, fairly easy to understand semantics and illustrates all issues salient for this article. It is slightly academic as we restrict the syntax far more than necessary to minimize the constructs to consider for the formal semantics of composition-free XQuery. However, many of the restrictions to the syntax can be dropped (e.g., we could integrate full navigational XPath as discussed in Section III-A) without affecting expressiveness and complexity, see also [20]. The only real restriction of composition-free XQuery in comparison to full XQuery is that it *disallows any querying of constructed nodes*, i.e., the domain of all relations is limited to the input nodes. This limitation clearly does not hold for full XQuery (even if we do not consider user-defined functions) and its effect on expressiveness and complexity is discussed in detail in [126].

(Composition-free) XQuery is built around controlled iterations over nodes of the input tree, expressed using **`for`** expressions. Controlled iteration is important for XQuery as it founded on sequences of nodes rather than sets of nodes (as XPath 1.0). In this respect it is more similar to languages such as DAPLEX [180] or OQL [48] than to XPath 1.0. (**`For`**) loops use XPath expressions for navigation and XML-look-a-likes for element construction all of which can be, essentially, freely nested. The following query gives an example of XQuery expressions. It creates a articlelist containing one author element for each author in the input XML tree (bound here and in the following to the canonical input variable `$inp`). For each such author, the nested **`for`** loop creates a list of all its articles. The latter expression can be more elegantly expressed in full XQuery using XPath qualifiers or **`where`** clauses but here it is shown in the "normalized" syntax of composition-free XQuery after [126].

```
<paperlist>
 for $a in $inp/descendant::author return
   <author> for $p in $inp/descendant::article return
     if some $x in $p/descendant::author satisfies
          deep-equal($x, $a)
     then $p
   </author>
</paperlist>
```

We choose to use deep-equal, XQuery's structural equality that tests whether the sub-trees at `$x` and `$a` are isomorphic, as authors can be represented using last and first name elements in our context and both have to be equal for it to be the same author.

A full definition of the syntax of composition-free XQuery is given in Table II. It deviates only marginally from [126] and [20]. In addition to the specification in Table II, the usual semantic restrictions apply, e.g., the label of the start and end tags must be the same, variables must be defined (using **`for`**) before use, etc. As stated, there is one exception from the latter, viz. the canonical input variable `$inp` which is always bound to the input XML tree.

In Table II, we use a general equality. XQuery provides in fact three kinds of equality, viz. node, atomic (or value), and deep equality. For all forms of equality the productions of Table II apply.

Again, compared to full XQuery the principle omission is the ability to query constructed nodes or values. In the syntax, this leads most prominently to the restriction of expressions following **`in`** in a **`for`**, i.e., expressions that provide bindings for variables, to XPath steps with variables. This way variables are always bound only to nodes

⟨*query*⟩	::=	⟨*query*⟩ ⟨*query*⟩ | ⟨*element*⟩ | ⟨*variable*⟩
	|	⟨*step*⟩ | ⟨*iteration*⟩ | ⟨*conditional*⟩
⟨*element*⟩	::=	'<' ⟨*label*⟩ '>' ⟨*query*⟩ '<' ⟨*/label*⟩ '>'
	|	'<' `lab(` ⟨*variable*⟩ `)>` ⟨*query*⟩ '</' `lab(` ⟨*variable*⟩ `)>`
⟨*step*⟩	::=	⟨*variable*⟩ '/' ⟨*axis*⟩ '::' ⟨*node-test*⟩
⟨*iteration*⟩	::=	`for` ⟨*variable*⟩ `in` ⟨*step*⟩ `return` ⟨*query*⟩
⟨*conditional*⟩	::=	`if` ⟨*condition*⟩ `then` ⟨*query*⟩
⟨*condition*⟩	::=	⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' '<' ⟨*label*⟩ '/>'
	|	`true`
	|	`some` ⟨*variable*⟩ `in` ⟨*step*⟩ `satisfies` ⟨*condition*⟩
	|	⟨*condition*⟩ `and` ⟨*condition*⟩
	|	⟨*condition*⟩ `or` ⟨*condition*⟩
	|	`not` ⟨*condition*⟩
⟨*axis*⟩	::=	`child` | `descendant` | `descendant-or-self`
	|	`next-sibling` | `following-sibling` | `following`
⟨*node-test*⟩	::=	⟨*label*⟩ | `node()`
⟨*variable*⟩	::=	'$'⟨*identifier*⟩

TABLE II
SYNTAX OF COMPOSITION-FREE XQUERY

from the input tree (anything reachable from `$inp` using XPath expressions). Another important omission is the absence of **`let`** clauses, which provide set-valued variables to XQuery. Conditional expressions are normalized to **`if`** clauses, where XQuery offers XPath qualifiers, **`where`** clauses, and **`if`** clauses.

Though **`order-by`** clauses are omitted, the result of an XQuery expression is always an ordered tree and the order of node construction must be precisely preserved (as given by the iteration of the **`for`** clauses which iterated over their respective node sequences mostly in document order).

j) Semantics: The semantics of a composition-free XQuery expression is then defined, following [20], using $[\![\,]\!]$ over a given such forest and a list of nodes from that forest $\vec{e} = [e_1, \ldots, e_n]$ that represent bindings for variables $x_1, \ldots, x_n$. For that, we assume that all variables are first renamed to x_i such that i is the number of variables in whose scope x_i is declared and assuming that `$inp` is scoped over the entire query. E.g., the query

```
for $x in $inp/child::a return
   for $y in $x/child::b return $x
for $z in $inp/child::c return
   for $v in $inp/child::d return $v
```

becomes

```
for $2 in $1/child::a return
   for $3 in $2/child::b return $2
for $2 in $1/child::c return
   for $3 in $1/child::d return $3
```

In the following, we assume that queries are in the latter form.

Table III specifies the semantics of composition-free XQuery on an XML forest F and a binding vector $\vec{e} = [e_1, \ldots, e_n]$ which is initially of length 1 containing bindings for `$inp`, i.e., usually one (or more, if querying XML collections) root node(s).

The semantics uses three auxiliary notions. 1) $\uplus$ is the union on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \uplus (F_2, \vec{e}_2) = (F_1 \cup F_2, \vec{e}_1 \circ \vec{e}_2)$ where $\circ$ is list (or vector) concatenation and the union of XML forests is defined component by component. 2) $\pitchfork$ is the intersection on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \pitchfork (F_2, \vec{e}_2) = (F_1, [e_i \in \vec{e}_1 : e_i \in \vec{e}_2])$. Note, that we only preserve F_1 (and thus $\pitchfork$ is *not* associative). However, for the purpose of the semantics the choice of the XML forest is arbitrary as $\pitchfork$ is only used for the semantics of conditions for which only the existence or non existence (and not their actual value) of bindings is relevant for the semantics of the full query. 3) $\mathsf{construct}(l, (F, [w_1, \ldots, w_n]))$ denotes construction of a new tree where l is a label, F is an XML forest and $[w_1, \ldots, w_n]$ is a vector of nodes in F. It returns a pair $(F \cup T', [\mathsf{root}(T')])$ where T' is a tree over a new set of nodes whose root $\mathsf{root}(T')$ is labeled with l and with the i-th subtree of $\mathsf{root}(T')$ isomorphic to the sub-tree rooted at w_i in F. Furthermore $\mathsf{construct}$ is assumed to return a tree with a distinct set of nodes each time it is called. This corresponds to *invention of new complex values* or trees.

Using these definitions, the semantics is fairly straightforward. In [20], Benedikt and Koch point out that most of the condition expressions (cases 10, 12–16) can be reduced to other XQuery expressions and thus do not need to be addressed in the semantics. We choose to give their definitions directly as the resulting expressions are no longer in composition-free XQuery.

The crucial parts of the semantics are cases 2 and 3, that illustrate element construction, case 7 that illustrates iteration, and case 8, the semantics of conditionals. The other cases are very similar to XPath and mostly just return appropriate binding vectors but leave F unchanged. Element construction (case 2 and 3) is achieved using the aforementioned $\mathsf{construct}$ function and returns a forest containing the newly constructed tree and bindings pointing to that tree's root node. Iteration using **`for`** has almost exactly the same semantics as the path separator `/` in XPath: the **`return`** expression is evaluated in the context of the **`in`** part, just like the subordinate path is evaluated in the context of the superordinate one. Indeed, the XQuery normalization transforms path expressions consisting of multiple steps to **`for`** loops as in composition-free XQuery. The difference is, of course, that the semantics of the **`return`** may be nodes from a newly constructed tree. It is crucial that this is the case *only* for the semantics of the **`return`** expression, not for that of the **`in`** expression which never modifies the given XML forest. In full XQuery, this does not hold, the **`in`** is followed by an arbitrary expression. Finally, conditionals are (again reminiscent of qualifiers in XPath) translated using a non-empty test on the bindings returned by the condition.

Note that the *relations of the input forest are never changed*. We may add new forests, but those do not have any relations to the input forest.

It is worth noting, that the semantics is uniform for boolean-valued conditions and for node-valued expressions

$$
\begin{array}{ll}
[\![\texttt{()}]\!](F,\vec{e}) & = (F,[]) \\
[\![\texttt{<}l\texttt{>}\, q \,\texttt{</}l\texttt{>}]\!](F,\vec{e}) & = \mathsf{construct}(l,[\![q]\!](F,\vec{e})) \\
[\![\texttt{<lab(\$}x_i\texttt{)>}\, q \,\texttt{</lab(\$}x_i\texttt{)>}]\!](F,[e_1,\ldots,e_n]) & \\
\qquad = \mathsf{construct}(\texttt{lab}(e_i),[\![q]\!](F,[e_1,\ldots,e_n])) & \\
[\![\texttt{\$}x_i]\!](F,[e_1,\ldots,e_n]) & = (F,[e_i]) \\
[\![\texttt{\$}x_i/axis\texttt{::}l]\!](F,[e_1,\ldots,e_n]) & = (F,[d : R_{axis}(e_i,d) \wedge \mathsf{Lab}^l(d)] \\
[\![q_1\ q_2]\!](F,\vec{e}) & = [\![query_1]\!](F,\vec{e}) \uplus [\![query_1]\!](F,\vec{e}) \\
[\![\texttt{for \$}x_i \texttt{ in } s \texttt{ return } q]\!](F,\vec{e}) & = \biguplus_{l \in \vec{l}} [\![q]\!](F,\vec{e}\cdot l) \text{ where } (F,\vec{l}) = [\![s]\!](F,\vec{e}) \\
[\![\texttt{if } cond \texttt{ then } q]\!](F,\vec{e}) & = \begin{cases} [\![query]\!](F,\vec{e}) & \text{if } \pi_2([\![cond]\!](F,\vec{e})) \neq [] \\ (F,[]) & \text{otherwise} \end{cases} \\
[\![\texttt{\$}x_i/axis\texttt{::node()}]\!](F,[e_1,\ldots,e_n]) & = (F,[d : R_{axis}(e_i,d)]) \\
[\![\texttt{some \$}x_i \texttt{ in } s \texttt{ satisfies } c]\!](F,\vec{e}) & = [\![\texttt{for \$}x_i \texttt{ in } s \texttt{ return } c]\!](F,\vec{e}) \\
[\![\texttt{\$}x_i = \texttt{\$}x_j]\!](F,[e_1,\ldots,e_n]) & = \begin{cases} (F,[e_i]) & \text{if } e_i = e_j \\ (F,[]) & \text{otherwise} \end{cases} \\
[\![\texttt{\$}x_i = \texttt{<}l\texttt{/>}]\!](F,[e_1,\ldots,e_n]) & = \begin{cases} (F,[e_i]) & \text{if = atomic equal and } \mathsf{Lab}^l(e_i) \\ (F,[e_i]) & \text{if = deep equal, } \mathsf{Lab}^l(e_i), \\ & \text{and } \nexists d : R_{\mathsf{child}}(e_i,d) \\ (F,[]) & \text{otherwise} \end{cases} \\
[\![c_1 \text{ or } c_2]\!](F,\vec{e}) & = [\![c_1]\!](F,\vec{e}) \uplus [\![c_2]\!](F,\vec{e}) \\
[\![c_1 \text{ and } c_2]\!](F,\vec{e}) & = [\![c_1]\!](F,\vec{e}) \pitchfork [\![c_2]\!](F,\vec{e}) \\
[\![\texttt{not } c]\!](F,\vec{e}) & = \begin{cases} (F,[\mathsf{root}(F)]) & \text{if } (F',[]) = [\![c]\!](F,\vec{E}) \\ (F,[]) & \text{otherwise} \end{cases} \\
[\![\texttt{true}]\!](F,\vec{e}) & = (F,[\mathsf{root}(F)])
\end{array}
$$

TABLE III
SEMANTICS FOR COMPOSITION-FREE XQUERY (FOLLOWING [20])

(in contrast to the XPath case in Section III-A). This follows [20] and allows a more compact definition of the semantics, at the cost of slightly surprising definitions for boolean operations and `true` in the latter part of the semantics. In the translation, we separate boolean-valued conditions from other expressions by a separate translation function as in the XPath case.

k) XQuery in industry and research: From the very start, XQuery's development has been followed by industry and research with equal interest (for reports on the challenges and decisions during this process see, e.g., [80], [83]). Even before the development has finished, initial practical introductions to XQuery have been published, e.g., [38], [120]. Industry interest is also visible in the simultaneous development of standardized XQuery APIs, e.g., for Java [81], and numerous implementations, both open source (e.g., Galax [86]) and commercial (BEA [87], IPSI-XQ [84]). Aside from these main-memory implementations, one can also find streamed implementations of XQuery (e.g., [17], [127]) where the data flows by as the query is evaluated. First results on implementing XQuery on top of standard relational databases (e.g., [73], [105]) indicate that this approach leads to very efficient query evaluation if a suitable relational encoding of the XML data is used. For more implementations, see the XQuery project page at the W3C and the proceedings of the first XIME-P workshop on "XQuery Implementation, Experience and Perspectives"[4].

It is intuitively clear that XQuery is Turing complete since it provides recursive functions and conditional expressions. A formal proof of the Turing-completeness of XQuery is given in [124]. Efficient processing and (algebraic) optimization of XQuery, although acknowledged as crucial topics, have not yet been sufficiently investigated. First results are presented, e.g., in [54], [57], [76], [145], [184], [199], [200]. Moreover, techniques for efficient XPath evaluation, as discussed above, can be a foundation for XQuery optimization.

Beyond querying XML data, it has also been suggested to use XQuery for data mining [189], for web service implementation [157], for querying heterogeneous *relational* databases [194], for access control and policy descriptions [151], for synopsis generation [65], and as the foundation of a visual XML query language (XQBE) [12], of a XML query language with full-text capabilities [7], [8], and of an update [39], [51], [168] and reactive [32] language for XML.

Recently, the W3C has proposed a revision [52] to XQuery 1.0, called XQuery 1.1, which among minor changes adds explicit grouping (using a new `group-by` clause) and iteration windows (or blockwise iteration, using a new `window` clause with several flavors). However, work on this revision is still in its infant stages.

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

C. XML-QL

To provide an alternative view on XML querying, we briefly glance at XML-QL [74], [75], a pattern- and rule-based query language for XML developed specifically to address the W3C's call for an XML query language (that resulted in the development of XQuery). It uses *query patterns* (called *element patterns* in [74]) in a `WHERE` clause. Such patterns can be augmented by variables for selecting data. The result of a query is specified as a *construction patterns* in the `CONSTRUCT` clause. An XML-QL query always consists of a single `WHERE-CONSTRUCT` rule, which may be divided into several (nested) subqueries.

The following listing shows an XML-QL query that selects all article elements in the bibliography and returns them together with a list of the last names of their authors in a result element.

```
WHERE
  <bib>
    <article>
    </article> ELEMENT_AS $b
  </bib>
CONSTRUCT
  <results>
    <result>
      $b
      WHERE <authors>
            <author>
              <last>$n</last>
            </author>
          </authors>
      CONSTRUCT <author>$n</author>
    </result>
  </results>
```

Variables are preceded in XML-QL by $. Note how the grouping of authors with their books is expressed using a nested query. In line 4, the variable `$b` is restricted to data matching the pattern in lines 3 and 4. Such "pattern restrictions" are indicated in XML-QL using the `ELEMENT_AS` keyword.

One of the main characteristics of XML-QL is that it uses query patterns containing multiple variables that may select several data items at a time instead of path selections that may only select one data item at a time. Furthermore, variables are similar to the variables of logic programming, i.e. "joins" can be evaluated over variable name equality. Since XML-QL does not allow one to use more than one separate rule, it is often necessary to employ subqueries to perform complex queries.

With the publishing of XQuery, XML-QL has not been further investigated. In particular, there are no results on complexity or expressiveness of XML-QL. Roughly speaking, the expressiveness is similar to that of non-compositional XQuery as discussed above.

D. Reachability in Trees

With XPath and XQuery as exemplars, we can broaden our attention to investigate how different query languages express reachability (or descendant and ancestor) queries in trees.

As XPath most XML query languages provide some form of path expression or axis for expressing different forms of reachability in a graph, most notably direct reachability or child axis vs. descendant axis. Path expressions have been introduced already for relational database, e.g., in GEM [197], an extension of QUEL, and for object-oriented databases, e.g., in OQL [48]). However, here path expressions require *fully specified* paths, i.e., paths with explicitly named nodes following only parent-child connections. OQL expresses paths with the "extended dot notation" introduced in GEM [197]: "`SELECT b.translator.name FROM Books b`" selects the name, or component, of the translator of books (note that there must be at most one translator per book for this expression to be legal).

Generalized (or regular) path expressions [59], [89], extend this notion with operators similar to regular expressions, e.g., the Kleene closure (and thus indirect reachability) operator on (sub-)paths but maintaining that each component is a node label. As a consequence and in contrast to the extended dot notation, generalized path expressions do not require explicit naming of all nodes along a path. Lorel [3] is an early exemplar of a semi-structured query language, yet based on a (graph-shaped) data model. Lorel's syntax resembles that of SQL and OQL, extending OQL's extended dot notation to generalized path expressions. To illustrate this aspect of Lorel, assume that one is only interested in books having "Julius Caesar" either as author or translator. Assume also that the literal giving the name of the author is either wrapped inside a `name` child of the `author` element, or directly included in the `author` element. Selecting only such books can be expressed in Lorel by a where clause filter on all books `B`: `where B.(author|translator).name? = "Julius Caesar"`.

Seeing that these efforts precede XPath significantly, it might seem surprising that XPath choose not to offer general path expressions but only the weaker axes. To recall, XPath allows navigation in *all directions* (vertical with descendant and ancestor, horizontal with following and preceding and their respective -siblings variants), while generalized path expressions only allow vertical navigation. However, it only provides closure axes (i.e., a path with any number of *arbitrarily* labeled nodes), but no closure of actual expressions. Thus it is, e.g., not possible to express that two elements are connected by only nodes with a certain label.

The first difference is clearly motivated by the particular emphasize placed on order in XML. However, the choice to provide only closure axes is less obvious. Without closure of arbitrary path expressions, XPath cannot express regular path expressions such as `a.(b.c)*.d` (meaning *"select d's that are reached via one a and then arbitrary many repetitions of one b followed by one c"*) and `a.b*.c`. Moreover it turns out that such a feature (called sometimes *conditional* axes) is exactly what is missing from XPath to become a first-order complete language on ordered trees [142], [143].

	time	space
Structural Joins, relational join	$\mathcal{O}(q \cdot n \cdot \log n)$	$\mathcal{O}(q \cdot n^2)$
————, *structure-aware join*	$\mathcal{O}(q \cdot n)$	$\mathcal{O}(q \cdot n^2)$
Twig or Stack Joins	$\mathcal{O}(q \cdot n)$	$\mathcal{O}(q \cdot n + n \cdot d)$
PDA-based (here: SPEX)	$\mathcal{O}(q \cdot n \cdot d)$	$\mathcal{O}(q \cdot n)$
Interval-based (here: CIQCAG)	$\mathcal{O}(q \cdot n)$	$\mathcal{O}(q \cdot n)$

TABLE IV
APPROACHES FOR XML TREE QUERY EVALUATION. n: NUMBER OF NODES IN THE DATA, d: DEPTH OF DATA; q: SIZE OF QUERY. WE ASSUME CONSTANT MEMBERSHIP TEST FOR ALL STRUCTURAL RELATIONS.

Moreover, the inclusion of reverse axes in XPath has been shown in [155] not to increase the expressive power of XPath. Consequently, they are used infrequently and, with the exception of the trivial parent axis, are considered *optional* features in XQuery that do not have to be provided by a conforming implementation.

Nevertheless, the efficient realisation of closure axes has proved to be one of the more fruitful issues on the road towards a scalable XML query language. In the following section, we classify approaches for implementing tree queries expressed in XPath or XQuery. All of these approaches have to deal in some form or the other with the presence of closure axes.

E. Tree Queries on Tree Data

Though XQuery (and even full XPath 1.0) can express also more powerful (graph) queries, the most significant results have been achieved on the implementation of tree queries (often roughly considered equivalent to navigational XPath, possibly without set operations and equality).

For tree queries, the restriction of XML to tree data can be exploited to provide highly efficient (linear time and space) evaluation of XML queries even in the absence of sophisticated indices.

To keep the discussion focused we ignore index-based evaluation of XML which is survey in [192]. Though path indices such as the DataGuide [95] or IndexFabric [67] and more recent variants [58] can significantly speed up path queries they suffer from two anomalies: First, if a tree query contains many branching nodes (i.e., nodes with more than one children) they generally do not perform better than, e.g., the structural join approach below. Second, even though only path queries can be directly answered from the index, the index size can be significantly higher than the size of the original XML documents.

We can classify most of the remaining approaches to the evaluation of XML tree queries in four classes (the corresponding complexity for evaluation XPath (and similar) tree queries on tree data is summarized in Table IV):

1) *Structural joins:* The first class is most reminiscent of query evaluation for relational queries and arguable inspired by earlier research on acyclic conjunctive queries on relational databases [100]. Tree queries are decomposed into a series of (structural) joins. Each structural join enforces one of the structural properties of the given query, e.g., a child or descendant relation between nodes or a certain label. Proposed first in [5], structural joins have also been used to great effect for studying the complexity of XPath evaluation and proposing the first polynomial evaluation of full XPath [99]. Due to its similarity with relational query evaluation it has proved to be an ideal foundation for implementing XPath and XQuery on top of relational databases [101]. It turns out, however, that the use of standard joins is often not an ideal choice and structure- or tree-aware joins [31] (that take into consideration, e.g., that only nodes in the sub-tree routed at another node can be that nodes a-descendants) can significantly improve XPath and XQuery evaluation.

2) *Twig joins:* In sharp contrast, the second class employs a single (thus called *holistic*) operator for solving an entire tree query rather than decomposing it into structural joins. These approaches are commonly referred to as twig or stack join [40], [56] and essentially operate by keeping one stack for each step in, e.g., an XPath query representing partial answers for the corresponding node-set. Theses stacks are organized hierarchically with (where possible, implicit) parent pointers connecting partial answers for upper stack entries to those of lowers. The approaches mostly vary in how the stacks are populated. In contrast to the other approaches, twig joins are limited to vertical, i.e., child and descendant, axes and have not been adapted for the full range of XPath axes. They also, like structure-aware joins [31], exploit the tree-shape of the data and can, at best, be adapted to DAGs [55].

3) *PDA-based:* Where twig joins assume one stream of nodes from the input document for each stack (and thus XPath step), the third class of approaches based on pushdown automata aims to evaluate XPath queries on a single input stream similar to a SAX event stream. SPEX, e.g., [152]–[154] also maintains a record of partial answers for each XPath step, but minimizes used memory more efficiently and exploits the existential nature of most XPath steps by maintaining only generic conditions rather than actual pointers to elements from the XML stream (except for candidates of the actual results set, of course). Also it supports all XPath axes in contrast to the twig join approaches. The cost is a slightly more complex algorithm.

4) *Interval-based*: Finally, interval-based approaches are a combination of the tree awareness in twig joins and SPEX and the structural join approach: The query is decomposed into a series of structural relations, but each relation is organised in such a way that all elements related to one element of its parent step are in a single continuous interval. This allows both an efficient storage and join of intermediate answers. The first interval-based approach are the Complete Answer Aggregates (CAA) [146], [147]. In

[90] the CIQCAG algebra is proposed which improves on the complexity of CAA (to the linear complexity given in Table IV) and covers, in contrast to CAA, arbitrary tree-shaped relations. It is also shown that interval-based approaches can be extended even to a large, efficiently detectable class of graph data (so called continuous-image graphs) that is not covered by any of the other linear time approaches discussed above.

Currently, extensions of the above algorithms for larger classes of graph data are investigated, e.g., in [55] and [90], see also Section IV-C on reachability in RDF graphs.

F. Supporting Order

In the previous sections, we have focused on the tree aspect of XML and its effect on query languages and their evaluation. However, XML is also set apart from many other data formats by an emphasize on ordered data that is very appropriate in a document setting such as XHTML or DocBook [188]. For query languages, which traditionally prefer a set-oriented perspective under the assumption that it enables more diverse evaluation strategies and thus better automatic optimization, this is a challenge that has been addressed in different ways in XML query languages.

Most of the early proposals ignore order in XML documents entirely or support it only superficially. Though XPath 1.0 allows querying the order, its results are either in document or in reverse document order, depending on the axis of the final step. This is fitting as XPath 1.0 is focused on selection and not (re-) construction of nodes.

For query languages like XQuery that also support construction of new XML trees, however, this is utterly insufficient. E.g., selecting authors together with their articles from the sample data in Section II-A and then constructing one XHTML section for each author containing a list of its articles requires control over the order in which section elements (e.g., h1s) and list elements (ul or ol) are intertwined.

This need is recognized in XQuery and, in many ways, all of XQuery is designed around proper support for ordered XML. Where in XPath 1.0 the results of path expressions are node sets, XQuery and XPath 2.0 use sequences. Sequences can be constructed or result from the evaluation of an XQuery expression. In contrast to XPath 1.0, sequences cannot only be composed of nodes but also from atomic values, e.g., `(1, 2, 3)` is a proper XQuery sequence. Combined with XQuery's iteration expression (for) we control precisely how we iterate both over nodes of the input and in which order we create new nodes. In this respect it is more similar to languages such as DAPLEX [180] which provide precise control over iteration on sequences of relational tuples, than to SQL, which only allows control over the order of the result sequence, let alone (set-based) relational algebra.

XSLT 2.0 [122] goes even further than XQuery in this respect. It is based on the same data model as XQuery (sequences of nodes), but also provides grouping based on order using the group-adjacent attribute of xsl:for-each-group.

The disadvantage of XQuery's (and XSLT 2.0's) choice to make order such prevalent in the language is that implementations have to painstakingly maintain this order to conform to the specification, see, e.g., [102] for a detailed account. In XQuery this has been partially recognized by providing the unordered keyword that allows a sub-query to be evaluated order indifferent, as if it had a set-based semantics. See, e.g., [103] on how to exploit order indifference in XQuery. Similarly, some alternative query languages, most notably Xcerpt [173] provide both ordered and unordered queries without preference for either.

This concludes our brief overview of XML query languages. For a comparison of a larger set of XML query languages see [14]. Here, we focus on highlighting some of the most innovative issues around XML query languages, viz. how languages cope with the need to query not only direct structural relations but also reachability, how the restriction on tree queries and tree data allows for a more efficient evaluation than on arbitrary relational data, and how order as a central concept in XML affects XML query languages. In all three cases, XML has triggered the development of novel approaches to query evaluation that have considerably extended our understanding of hierarchical queries in general. In the next section, we turn to RDF and try to illustrate where similar questions arise for RDF querying, though RDF being a considerably less established data format and topic of research shows in the comparative lack of significant advances to existing knowledge about query evaluation.

IV. Queries as Programs II: RDF Query Languages

Compared with XML query languages, the field of RDF query languages is less mature and has not received as much attention from research, just as RDF in general. Recently, the W3C has started to derive a standard RDF query language, called SPARQL [165], that is, visibly influenced by languages such as RDQL [149], RQL [119], and SeRQL [37], aiming to create a stable foundation for use, implementation, and research on RDF databases and query languages. Where XML query languages focus on trees and order, RDF query languages have to deal with the simple, but also highly flexible RDF: RDF data comes (see Section II-B) in the shape of arbitrary (usually node- and edge-labeled) graphs. Yet surprisingly and in stark contrast to the XML case, many RDF query languages only provide access to direct properties, but not to reachability information, see Section IV-C. In contrast to relational or object-oriented (which can also be considered representing graph data) data *all* properties (i.e., outgoing edges) are *optional* and *multi-valued*. For instance, an author may or may not have a last name and may even have many such names. How query languages deal with this inherent optionality is discussed in Section IV-D. Resources (i.e., nodes) are in general labeled with (globally) unique identifiers that allow

us to talk about the same resource in different data sets. However, RDF also allows blank nodes which play the role of local-only identifiers. Blank nodes are like existential data and pose particular challenges for RDF query evaluation (see Section IV-E).

Again, we start off the discussion of RDF query languages with a closer look at two of the more prominent exemplars: SPARQL and RQL. These introductions are focused on their essentials. For a more in-depth comparison of (more than a dozen) RDF query languages see [92].

A. SPARQL 1000 Words

Fundamentally, SPARQL is a fairly simple query language in the spirit of basic subsets of SQL or OQL. However, the specifics of RDF have lead to a number of unusual features that, arguably, make SPARQL more suited to RDF querying than previous approaches such as RDQL [149]. However, the price is a more involved semantics complemented by a tendency in [165] to redefine or ignore established notions from relational and XML query languages rather than build upon them.

Nevertheless, SPARQL is expected to become the "lingua franca" of RDF querying and thus well worth further investigation. In the following sections, we first briefly introduce into SPARQL and its semantics (based on [161] and [162] but extended to full SPARQL queries rather than only patterns).

l) Example.: The following SPARQL query selects from the graph in Section II-B all articles in the journal with name "Computer Journal" and returns a new graph where the bib:isPartOf relation of the original graph is inverted to bib:hasPart.[5]

```
CONSTRUCT { ?j bib:hasPart ?a }
WHERE { ?a rdf:type bib:Article AND ?a bib:isPartOf ?j
        AND ?j bib:name 'Computer Journal' }
```

The query illustrates SPARQLs fundamental query construct: a pattern (s, p, o) for RDF triples (whose components are usually thought of as subject, predicate, object). Any RDF triple is also a triple pattern, but triple patterns allow variables for each component. Furthermore, SPARQL also allows literals in subject position, anticipating the same change also in RDF itself. We use the variant syntax for SPARQL discussed in [161] to ease the definition of syntax and semantics of the language. For instance, standard SPARQL, uses . instead of AND for triple conjunction. We consider two forms of SPARQL queries, viz. **SELECT** queries that return list of variable bindings and **CONSTRUCT** queries that return new RDF graphs. Triple patterns contained in a **CONSTRUCT** clause (or "template") are instantiated with the variable bindings provided by the evaluation of the triple pattern in the **WHERE** clause. We omit named graphs and assume that all queries are on the single input graph. An extension of the discussion to named graphs is easy (and partially demonstrated in [162]) but only distracts from the salient points of the discussion.

The full grammar of SPARQL queries as considered here (extending [161] by **CONSTRUCT** queries) is as follows:

⟨*query*⟩ ::= 'CONSTRUCT' ⟨*template*⟩ 'WHERE' ⟨*pattern*⟩
| 'SELECT' ⟨*variable*⟩+ 'WHERE' ⟨*pattern*⟩
⟨*template*⟩ ::= ⟨*triple*⟩ | ⟨*template*⟩ 'AND' ⟨*template*⟩ | '{' template '}'
⟨*triple*⟩ ::= ⟨*resource*⟩',' ⟨*predicate*⟩',' ⟨*resource*⟩
⟨*resource*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*blank*⟩
⟨*predicate*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩
⟨*variable*⟩ ::= '?' ⟨*identifier*⟩
⟨*pattern*⟩ ::= ⟨*triple*⟩ | '{' ⟨*pattern*⟩ '}'
| ⟨*pattern*⟩ 'FILTER' '(' ⟨*condition*⟩ ')' |
| ⟨*pattern*⟩ 'AND' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'UNION' ⟨*pattern*⟩
| ⟨*pattern*⟩ 'MINUS' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'OPT' ⟨*pattern*⟩
⟨*condition*⟩ ::= ⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' (⟨*literal*⟩|⟨*iri*⟩)
| 'BOUND(' ⟨*variable*⟩ ')' | 'isBLANK(' ⟨*variable*⟩ ')'
| 'isLITERAL(' ⟨*variable*⟩ ')' | 'isIRI(' ⟨*variable*⟩ ')'
| ⟨*negation*⟩ | ⟨*conjunction*⟩ | ⟨*disjunction*⟩
⟨*negation*⟩ ::= '¬'⟨*condition*⟩
⟨*conjunction*⟩ ::= ⟨*condition*⟩ '∧' ⟨*condition*⟩
⟨*disjunction*⟩ ::= ⟨*condition*⟩ '∨' ⟨*condition*⟩

We pose some additional syntactic restrictions: SPARQL queries are *range-restricted*, i.e., all variables in the "head" (**CONSTRUCT** or **SELECT** clause) also occurs in the "body" (**WHERE** clause) of the query. We assume *error-free* SPARQL expressions (in contrast to [161] and [162]), i.e., for each **FILTER** expression all variables occurring in the (right-hand) condition must also occur in the (left-hand) pattern. The first limitation is as in standard SPARQL, the second is allowed in standard SPARQL but can easily recognized a-priori and rewritten to the canonical false **FILTER** expression (as **FILTER** expressions with unbound variables raise errors which, in turn, are treated as a false filter, see "effective boolean value" in [165].

Finally, we allow only *valid RDF constructions* in **CONSTRUCT** clauses, i.e., no literal may occur as a subject, all variables occurring in subject position are never bound to literals, and all variables occurring in predicate position are only ever bound to IRIs (but not to literals or blank nodes). The first condition can be enforced statically, the others by adding appropriate isIRI or negated isLITERAL filters to the query body.

Following [162], we define the semantics of SPARQL queries based on *substitutions*. A substitution $\theta = \langle v_1, n_1, \ldots, v_k : n_k \rangle$ with $v_i \in \mathsf{Vars}(Q) \wedge n_i \in nodes(D)\}$ for a query Q over a data graph D maps some variables from Q to nodes in D. For a substitution θ we denote with $\mathrm{dom}(\theta)$ the

[5] Here, and in the following we use namespace prefixes to abbreviate IRIs. The usual IRIs are assumed for rdf, rdfs, dc (dublin core), foaf (friend-of-a-friend), vcard vocabularies. bib is a prefix bound to an arbitrary IRI.

variables mapped by θ. Given a triple pattern $t = (s, p, o)$, we denote with $t\theta$ the application of θ to t replacing all occurrences of variables mapped in θ by their mapping in t. For a triple (s, p, o) containing no variables, we say $(s, p, o) \in D$ if there is a p labeled edge between s and o labeled nodes in D.

On sets of substitutions the usual relational operations $\bowtie$, $\cup$, and $\setminus$ apply. We define the (left) semi-join $R \ltimes S = (R \bowtie S) \cup (R \setminus S)$.

Finally, given a template t, i.e., a conjunction of triple patterns, $\mathsf{std}(t)$ returns t but replacing each blank node identifier (i.e., strings of the form _:*identifier*) with a new blank node identifier not occurring in D and not created by a prior application of std. Intuitively, $\mathsf{std}(t)$ creates a new instance of t such that the blank nodes of two instances (and any instance with the input graph) do not overlap.

Using these definitions, Table V gives the semantics of SPARQL **SELECT** and **CONSTRUCT** queries by means of $[\![\,]\!]^D$. $[\![\,]\!]^D$ translates the **WHERE** clause using $[\![\,]\!]^D_{\mathsf{Subst}}$ and a **CONSTRUCT** clause, if present, using $[\![\,]\!]^D_{\mathsf{Graph}}$. For a **SELECT** query, we project the set of substitutions returned by $[\![\,]\!]^D_{\mathsf{Subst}}$ to the set of answer variables V. For a **CONSTRUCT** query we apply each substitution $\theta \in [\![P]\!]^D_{\mathsf{Subst}}$ to a new instance of the template t contained in the **CONSTRUCT** clause created using std. Applying a substitutions is straightforward except that triples containing one or more variables that bound to **nil** by θ are omitted entirely.

The semantics of a SPARQL pattern P contained in the **WHERE** clause is given by $[\![P]\!]^D$ and produces a set of substitutions (or bindings) for variables in P. Triple patterns t (case 1) are evaluated to the set of substitutions θ such that the $t\theta$ contains no more variables and falls in D. Pattern compositions **AND**, **UNION**, **MINUS**, and **OPT** are reduced to the appropriate operations on sets of substitutions (cases 2–4). **FILTER** expressions (case 5) are again evaluated straightforwardly, as restrictions on the substitutions returned by the (left-hand) pattern with the boolean formula that is provided by $[\![\,]\!]^D_{\mathsf{Bool}}$ for the condition of the filter expression. $\mathsf{Vars}(\mathit{condition}) \subset \mathrm{dom}(\theta)$ is not strictly necessary as it merely restates that we only consider error-free SPARQL queries.

Recently, SPARQL has been the target of a number of studies and extensions. Its complexity and formal semantics have been studied in [161], where it is shown, that, unsurprisingly, full SPARQL patterns are just as expressive as relational algebra and thus PSPACE-complete w.r.t. query complexity. This is somewhat disappointing as thus many graph queries (including simple reachability queries) are not expressible in SPARQL, yet highly desirable for RDF query languages, see, e.g., [10]. Extensions of SPARQL with rules [162], [175] have received some attention in part as they can address some of these weaknesses and as they are seen as the natural next step towards a Semantic Web query engine. Also studied have been several embeddings of SPARQL in XQuery or vice versa, see, e.g., [163].

B. RQL and SeRQL

Under "RQL family", we group the languages *RQL* [119] and *SeRQL* [37]. 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 [60], [116]–[119]. 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 (see also Section VI). 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*".[6] The resulting language is, unsurprisingly, of rather limited expressiveness and can not express most of the sample queries.

m) Basic schema queries.: A salient feature of RQL is the use of the types from RDFS schemas. The query subClassOf(bib:Article) returns the sub-classes of the class bib:Article. 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:

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

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

$$
\begin{aligned}
&[\![(s,p,o)]\!]^D_{\text{Subst}} &&= \{\theta : \text{dom}(\theta) = \text{Vars}((s,p,o)) \wedge t\theta \in D\}\\
&[\![pattern_1 \texttt{ AND } pattern_2]\!]^D_{\text{Subst}} &&= [\![pattern_1]\!]^D_{\text{Subst}} \bowtie [\![pattern_2]\!]^D_{\text{Subst}}\\
&[\![pattern_1 \texttt{ UNION } pattern_2]\!]^D_{\text{Subst}} &&= [\![pattern_1]\!]^D_{\text{Subst}} \cup [\![pattern_2]\!]^D_{\text{Subst}}\\
&[\![pattern_1 \texttt{ MINUS } pattern_2]\!]^D_{\text{Subst}} &&= [\![pattern_1]\!]^D_{\text{Subst}} \setminus [\![pattern_2]\!]^D_{\text{Subst}}\\
&[\![pattern_1 \texttt{ OPT } pattern_2]\!]^D_{\text{Subst}} &&= [\![pattern_1]\!]^D_{\text{Subst}} \ltimes [\![pattern_2]\!]^D_{\text{Subst}}\\
&[\![pattern \texttt{ FILTER } condition]\!]^D_{\text{Subst}} &&= \{\theta \in [\![pattern]\!]^D_{\text{Subst}} : \text{Vars}(condition) \subset \text{dom}(\theta)\\
&&&\qquad \wedge [\![condition]\!]^D_{\text{Bool}}(\theta)\}\\
\\
&[\![condition_1 \wedge condition_2]\!]^D_{\text{Bool}}(\theta) &&= [\![condition_1]\!]^D_{\text{Bool}}(\theta) \wedge [\![condition_2]\!]^D_{\text{Bool}}(\theta)\\
&[\![condition_1 \vee condition_2]\!]^D_{\text{Bool}}(\theta) &&= [\![condition_1]\!]^D_{\text{Bool}}(\theta) \vee [\![condition_2]\!]^D_{\text{Bool}}(\theta)\\
&[\![\neg condition]\!]^D_{\text{Bool}}(\theta) &&= \neg [\![condition]\!]^D_{\text{Bool}}(\theta)\\
&[\![\texttt{BOUND}(?v)]\!]^D_{\text{Bool}}(\theta) &&= v\theta \neq \texttt{nil}\\
&[\![\texttt{isLITERAL}(?v)]\!]^D_{\text{Bool}}(\theta) &&= v\theta \in L\\
&[\![\texttt{isIRI}(?v)]\!]^D_{\text{Bool}}(\theta) &&= v\theta \in I\\
&[\![\texttt{isBLANK}(?v)]\!]^D_{\text{Bool}}(\theta) &&= v\theta \in B\\
&[\![?v = literal]\!]^D_{\text{Bool}}(\theta) &&= v\theta = literal\\
&[\![?u = ?v]\!]^D_{\text{Bool}}(\theta) &&= u\theta = v\theta \wedge u\theta \neq \texttt{nil}\\
\\
&[\![triple]\!]^D_{\text{Graph}}(\theta) &&= triple\theta \text{ if } \forall v \in \text{Vars}(triple) : v\theta \neq \texttt{nil},\ \top \text{ otherwise}\\
&[\![template_1 \texttt{ AND } template_2]\!]^D_{\text{Graph}}(\theta) &&= [\![template_1]\!]^D_{\text{Graph}}(\theta) \cup [\![template_2]\!]^D_{\text{Graph}}(\theta)\\
&[\![\texttt{CONSTRUCT } t \texttt{ WHERE } p]\!]^D &&= \bigcup_{\theta \in [\![P]\!]^D_{\text{Subst}}} [\![\text{std}(t)]\!]^D_{\text{Graph}}(\theta)\\
&[\![\texttt{SELECT } V \texttt{ WHERE } p]\!]^D &&= \pi_V([\![P]\!]^D_{\text{Subst}})
\end{aligned}
$$

TABLE V
SEMANTICS FOR SPARQL

2) using a *type constraint*[7]:

```
SELECT C1, C2 FROM Class{C1}, Class{C2},
    {;C1}bib:author{;C2}
```

3) without class variables or type constraints:

```
SELECT C1, C2 FROM
    subClassOf(domain(bib:author)){C1},
               subClassOf(range(bib:author)){C2}
```

While the first two queries return exactly the same result—namely the domain and range of the bib:author-property and all possible combinations of their subclasses—the third query does not include the domain and range of bib: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(bib:Article) returns the top of the subsumption hierarchy that Article is part of. 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 bib:Article:

```
SELECT @P, $V FROM {;bib:Article}@P{$V}
```

n) 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 `bib:Article` returns the resources classified as bib:Article or as one of its sub-classes. This query can also be expressed as follows: `SELECT X FROM bib:Article{X}`. Prefixing the variable `X` with `^` in the previous queries, yields queries returning only resources directly classified as bib:Article, i.e., for which a statement $(X, \text{rdf:type}, \text{bib:Article})$ exists. The extent of a property can be similarly retrieved. The query `^bib:author` returns the pairs of resources X, Y that are in the bib:author relation, i.e., for which a statement $(X, \text{bib:author}, Y)$ exists. RQL offers extended dot notation as used in OQL [48], for navigation in data and schema graphs. The data selected by an RDF query can be restricted with a WHERE clause:

```
SELECT X, Y FROM {X;bib:Article}bib:isPartOf.bib:name{Y},
                 {X}dc:title{T}
WHERE T = "...Semantic Web..."
```

[7] In the following we omit the namespace part.

o) Mixed schema and data queries.: With RQL, access to data and schema can be combined in all manners, e.g., the expression `X;bib:Article` restricts bindings for variable `X` to resources with type bib:Article. 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}dc:title{T} WHERE T = "...Semantic
     Web..."
```

This query returns the classes under which the resource with title "...Semantic Web..." 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 [137] 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 capable of expressing a wider range of queries than most other RDF query languages, especially those of the SPARQL family.

For a detailed formal semantics for RQL see [119].

C. Reachability

In stark contrast to the XML case, many RDF query languages do not provide means to access reachability information or any other form of navigation in the RDF graph beyond direct edge traversal. In [10], a set of graph queries that are desirable for an RDF query language are described, but neither SPARQL nor RQL can express the majority of these constructs.

However, if we look beyond SPARQL and RQL we find that RDF query languages actually support a wide variety of path expressions:

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 [197], OQL [48], SPARQL [165], and RQL [119].

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 [74], Quilt [53], XPath and all XML query languages based on XPath). It is also used in the RDF query language iTQL [1]. 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 [143]. 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 [142] 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:

p) Evaluation of reachability queries on graphs.: For tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [78] or other labeling schemes such as [193]). However, for graph data this is not so obvious. Fortunately, there has been considerable research on reachability or closure relations and their indexing in arbitrary graph data in recent years. Table VI summarizes the most significant approaches for RDF querying.

Obviously, we can obtain constant time for the member-

ship test if we store the full transitive closure matrix. However, for large graphs this is clearly infeasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test.

The first class are based on the idea of a 2-hop cover [62]: Instead of storing a full transitive closure, we allow that reachable nodes are reached via at most one other node (i.e., in two "hops"). More precisely, each node n is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach n, $out(n)$ a set of nodes that are reachable from n. Both sets are assigned in such a way, that a node m is reachable from n iff $out(n) \cup in(m) \neq \emptyset$. Unfortunately, computing the optimal 2-hop cover is NP-hard and even advanced approximation algorithms [176] have still rather high complexity.

A different approach [4], [55], [185], [190] is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [190] where a full transitive closure over the non-tree edges is build. GRIPP [185] and SSPI [55] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

This consideration shows that reachability (at least in its basic form) does not need to significantly degrade the performance of RDF query evaluation (and clearly does not affect its complexity, seeing that already SPARQL SELECT queries are PSPACE-complete).

D. Optionality

So far, we have mostly focused on 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 articles and editors for articles that have editors and just articles otherwise. If one considers the results of a query as a table with **nil** values, the translator column is **nil** in the latter case, i.e., if there is no bib:editor property for that article.

```
SELECT ?article, ?editor
WHERE { ?article a bib:Article AND
        OPTIONAL { ?article bib:editor ?editor } }
```

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 ?article, ?editor
WHERE { ?article a bib:Article AND
          ?article bib:editor ?editor }
        UNION
        { ?article a bib:Article }
```

Furthermore, the latter is not actually equivalent as it returns also for writings X with translators one result tuple (X,**nil**). 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 $A \wedge \text{optional}(B) \wedge \text{optional}(C)$ 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 inter-dependent 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 articles together with editors and, if that editor is also an author, also the author name.

```
SELECT ?article, ?person, ?name
WHERE { ?article a bib:Article AND
        OPTIONAL { ?article bib:editor ?person }
        OPTIONAL { ?article bib:author ?person AND
                ?person bib:name ?name } }
```

If we change the order of the two optional parts, the semantics of the query changes: select all articles 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 an editor 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 ?article, ?person, ?name
WHERE { ?article a bib:Article .
        OPTIONAL { ?article bib:editor ?person
          OPTIONAL { ?article bib:author ?person AND
                  ?person bib:name ?name }
        } }
```

approach	characteristics	query time	index time	index size
Shortest path [159]	no index	$\mathcal{O}(n+e)$	–	–
Transitive closure	full reachability matrix	$\mathcal{O}(1)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
2-Hop [62]	2-hop cover[a]	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n \cdot \sqrt{e})$
HOPI [176]	2-hop cover, improved approximation algorithm	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n \cdot \sqrt{e})$
Graph labeling [4]	interval-based tree labeling and propagation of intervals of non-tree descendants.	$\mathcal{O}(n)^b$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)^c$
SSPI [55]	interval-based tree labeling and recursive traversal of non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$
Dual labeling [190]	interval-based tree labeling and transitive closure over non-tree edges	$\mathcal{O}(1)^d$	$\mathcal{O}(n+e+e_g^3)$	$\mathcal{O}(n+e_g^2)$
GRIPP [185]	interval-based tree labeling plus additional interval labels for edges with incoming non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$

[a] Index time for approximation algorithm in [62].

[b] More precisely, the number of intervals per node. E.g., in a bipartite graph this can be up to n, but in most (sparse) graphs this is likely considerably lower than n.

[c] More precisely, the total number of interval labels.

[d] [190] introduces also a variant of dual labeling with $\mathcal{O}(\log e_g)$ query time using a, in practical cases, considerably smaller index. However, worst case index size remains unchanged.

TABLE VI
COST OF MEMBERSHIP TEST FOR CLOSURE RELATIONS. n, e: NUMBER OF NODES, EDGES IN THE DATA, e_g: NUMBER OF NON-TREE EDGES, I.E., IF $T(D)$ IS A SPANNING TREE FOR D WITH EDGES $E_{T(D)}$, THEN $e_g = |E_D \setminus E_{T(D)}|$.

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.

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: $A \wedge \text{optional}(B) \wedge \text{optional}(C)$ is equivalent to $(A \wedge \text{not}(B) \wedge \text{not}(C)) \vee (A \wedge \text{not}(B) \wedge C) \vee (A \wedge B \wedge \text{not}(C) \vee (A \wedge B \wedge C)$. This semantics ensures that the maximal number of optionals for a certain (partial) variable assignment is used. This semantics has been introduced in Xcerpt [173].

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.

E. Existential Information

Recall, that RDF data may contain specifically marked resources (called *blank nodes*) that remain without identity but express only existential information. In fact, if we consider an RDF graph as a logical conjunction of triples they become existential quantifiers over the resulting formula. They pose a number of challenges for RDF query evaluation.

First, when blank nodes are selected by a query should a query language return them like any other resource? Recall, that blank nodes are essentially local identifiers and thus outside the scope of their original graph may not carry much information. Furthermore, blank nodes express existential information and such information may be redundant, i.e., already implied by the other data. E.g., if the data contains the statement that the article smith2005 is part of issue 11 of some journal in addition to the data from Figure 2 that information is obviously implied already from the remaining data (that smith2005 is part of the issue 11 of the journal "Computer Journal") and thus can be safely omitted. An RDF graph without such redundant information is called *lean* [109]. Ideally, we might expect an RDF query language to return only blank nodes that are non-redundant (and for these maybe enough additional information to retrieve them again, e.g., a concise bounded description [182]). However, simply computing the lean graph for any given RDF graph is already CO-NP-complete [107] and thus undesirable for most query evaluation. Thus most RDF query languages choose to ignore this issue and return blank nodes just like any other resource.

Second, when constructing new RDF graphs (e.g., through SPARQL's `CONSTRUCT` clause) we need to be able to construct also new blank nodes to obtain an adequate RDF query language. However, such blank node construction

easily introduces a form of construction: Say we want to construct a new blank node with edges to all articles selected by this query. Then a single blank node for all articles is needed. However, we might also want to construct, for each article, a new blank node with edges to each of its authors. Now we need one "fresh" blank node for each article (otherwise all articles share all authors) but only one for each group of authors of the same article. SPARQL only allows the construction of blank nodes that are in the scope of *all* query variables and thus can express neither of the above queries. In RDFLog [44], [45] the effect of blank nodes on RDF querying is studied in detail. It is shown, in particular, that the combination of blank node support (even as in SPARQL) with (recursive) rules (as, e.g., in [175]) immediately leads to an undecidable, Turing-complete language that can be reduced, using Skolemization and a novel form of un-Skolemization, to standard logic programming. It is also shown that arbitrary scoping of blank nodes is not more expensive as SPARQL-style $\forall\exists$ scopes and that, at least in presence of rules, the two are actually equivalent.

This concludes our brief summary of core issues on RDF querying and RDF query languages. For a comparison of a larger set of RDF query languages see [92]. The above discussion shows clearly that RDF querying is yet a less mature field of research than the XML case, but that there are a number of open questions that urgently need to be addressed for efficient and convenient access to RDF, and thus arguably the entire Semantic Web vision to move forward.

V. Queries as Programs—Outlook: Versatile Languages

In the previous sections we have separated Web and Semantic Web query languages into XML and RDF query languages. However, in recent years GRDDL [66] and similar initiatives have given renewed evidence to the effort of defining a means to conveniently access both XML and RDF data within the same application and even same query language. This has been reflected in an increasing number of integration approaches for XML and RDF querying. Previous approaches for integrating XML and RDF access fall roughly into two categories: transformation and multi-language approaches. In the former, a pure XML or a pure RDF query language is used and data in the respectively other format can only be accessed by some encoding in the other one. In the latter, a query language for one of the data formats is combined, most often embedded into the other one (e.g., XSPARQL, GRDDL). The advantage of transformation approaches is that users have to learn only a single language. However, this is offset by the need to understand the encoding of RDF in XML or vice versa and very limited support for specifics of the encoded data format that are not present in the native format.

A unique position among these approaches is held by Xcerpt and its extension Xcerpt^{RDF}: Through a slight extensions to the pattern- and rule-based XML query language Xcerpt convenient querying of RDF is enabled that, in contrast to, e.g., SPARQL, address also the graph nature of RDF. The vast majority of language features is shared by both the XML and the RDF version of Xcerpt, thus alleviating the problems of the above mentioned integration approaches.

For a more detailed look at Xcerpt see [173] and for its RDF extension Xcerpt^{RDF} [46].

Here, we briefly outline the basic ideas of Xcerpt to give an impression of how a versatile semi-structure query language compares with XQuery or SPARQL as discussed in the previous sections.

A. Xcerpt

Xcerpt [173] is a query language designed after principles given in [42] for querying both data on the "standard Web" (e.g., XML and HTML data) and data on the Semantic Web (e.g., RDF, Topic Maps, etc. data). Xcerpt is "data versatile", i.e. the same Xcerpt query can access and generate, as answers, data in different Web formats. Xcerpt is "strongly answer-closed", i.e. it not only allows one to construct answers in the same data formats as the data queries like, e.g., XQuery [50], but also allows further processing of the data generated by this same query program. Xcerpt's queries are pattern-based and allow to incompletely specify the data to retrieve, by (1) not explicitly specifying all children of an element, (2) specifying descendant elements at indefinite depths (restrictions in the form of regular path expressions being possible), and (3) specifying optional query parts. Xcerpt's evaluation of incomplete queries is based on a novel unification algorithm called "simulation unification". Xcerpt's processing of XML documents is graph-oriented, i.e., Xcerpt is aware of the reference mechanisms (e.g., ID/IDREF attributes and links) of XML.

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing, data from existing data (i.e. the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of programs.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs, viz. Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals"] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by allowing unordered data and making references first class citizens (thus moving from a tree to a proper graph data model).

For instance, in the following query the construct rule defines data about books and their authors which is then queried by the goal. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) holds.

A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
  authors [ var X ]
FROM
  book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ],
        author [ "William Dalrymple" ] ] END
```

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique[8] to match data terms. Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching (indicated by different types of brackets). Query terms may also contain (logic) variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. Matching, for instance, against the XML snippet above the query term `book [ title [ var X ] ]` with results in the variable binding `{X/"White Mughals"}`. In addition to the query term types discussed in [173], we also consider non-injective ordered and unordered query terms indicated by three braces or brackets, respectively.

Construct terms are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the example above we obtain the data term `authors [ "William Dalrymple"]` as result.

A visual language, called *visXcerpt* [24], [25], 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.

More details on Xcerpt and the vision of versatile Web querying can be found in [42], [43], [90] and on its implementation in [41], [90].

VI. Queries as Keywords: Keyword-based Query Languages

In the literature, the term *keyword-based query language* is used to refer to a group of query languages that use (relatively) unstructured bags of words that the user deems significant as queries or a part thereof. A typical characteristic of keyword query languages is the implicit conjunctive semantics, that is, by default the data must contain all words in a query to be a match without this being explicitly expressed in the query.

The traditional query languages for semi-structured data discussed in the introduction require at least some knowledge of the structural organization of the data to be queried as well as of the syntactic constructs and principles of the query language. In contrast to this, the motivation behind developing keyword-based query languages for XML and RDF is to enable casual users to construct queries and obtain useful results without having to undergo training in a query language, having to know the underlying structure of the data and even without having a clear understanding of the data structures. Another advantage of these query languages is that they allow for querying over heterogeneous data i.e. data with different underlying schemas.

Keyword queries are not only easy to construct, but have proven to be surprisingly effective in helping the user to localize relevant information. Consequently, keyword querying has become an established technique for finding information that virtually all Web users are familiar with. Keyword search is used in a wide variety of applications and domains, in Web search engines such as Google[9] and Yahoo![10] which allow for general internet search over various types of documents as well as in more specialized contexts and domains. For example, the query "XML Web" entered into Google yields a lists of Web pages in which the terms occur. On the shopping site Amazon[11] and the auction site Ebay[12] it results in a list of products available on the site and on the social networking site Facebook[13], the search results for the same query contain relevant user groups, events, user profile add-ons and users who are interested in the Web and XML.

Since keyword-based querying is established and shows great effectiveness in querying the Web in a variety of domains, it is a promising and worthwhile approach to achieve non-expert querying of XML and RDF data. Various keyword query languages have been proposed in recent years. The goal of this section is to present the different approaches taken and to give an overview over the different paradigms and concepts as well as the expressive power of the individual languages. Keyword querying in relational databases (see e.g. [112]) is a related topic that will not be treated here.

A. Characteristics of keyword query languages

Keyword querying as normally used on the Web on the one hand and traditional RDF and XML query languages on the other hand can be seen as two extremes with respect to the degree to which querying the structure of the data is possible; in the former, the structure of the data cannot normally be queried. For example, Amazon's regular search interface does not allow to indicate that a keyword should be matched on authors' names. In the latter case, the structure of the data may (but does not necessarily have to be, see below) fully specified, for example in SPARQL and XPath.

[8] Called *simulation unification*. For details of this technique, please refer to [172].

[9] http://www.google.com/
[10] http://www.yahoo.com/
[11] http://www.amazon.com/
[12] http://www.ebay.com/
[13] http://www.facebook.com/

Based on this observation, at least three different types of keyword-based query languages for structured data can be distinguished according to where they fall on this spectrum:

1) *Keyword-only query languages*, the simplest variant. Here, queries consist of a number of words, which are usually matched on the textual content of nodes in an XML or RDF document. In some cases, the keywords may also be matched against node or, in the case of RDF, edge labels, but generally, the query makes no reference to the structure of the data.
   Most keyword query languages for XML and RDF presented in the following fall into this category, for example XKeyword [15], [111], XRank [106] and XKSearch [196].
2) *Label-keyword query languages*, e.g. XSearch [64], where a query term is a label-keyword pair of the form *l:k*. The term matches data where a node with the label *l* has textual content, either directly or through a descendant node, matching the associated keyword *k*. It is thus possible to indicate some structure in the query.
   Depending on the query language, either the label or the keyword may be optional, meaning that query terms may be of the form *:k*, *l:* or *l:k*. For example, the query "title:Web" applied to the example data, an excerpt from a fictional bibliographical XML database (Figure 1) matches nodes 3 and 17. The query ":Web" on the other hand, a keyword-only query since no label is specified, matches nodes 3, 17 and 26, since it does not impose constraints on the node label.
   The difference between keyword-only languages that match both on labels and content and label-keyword languages is that in the latter case, the association between label and keyword and the kind of element each keyword refers to (node label or node content) is represented, allowing for more targeted queries since label-content or parent label-descendant content relationships can be explicitly specified.
3) Keyword-enhanced query languages like for instance Schema-Free Query [133] integrate traditional query languages like XPath with simple keyword querying as described in this article. They allow for the specification of the structure to the degree that it is known, but also include constructs that make it possible to use keyword-querying where it is not, thus offering a smooth transition between keyword querying and creating fully specified queries.
   Keyword-enhanced query languages constitute extensions of traditional query languages, meaning that they provide their full power.
   Since traditional query languages offer ways to specify queries when the user lacks knowledge about the data structure, e.g. through regular path expressions in XPath, the question arises how traditional query languages and Keyword-enhanced query languages can be differentiated. As is pointed out in [88] and [178], regular path expressions are useful if the the structure is not completely known to the user, but are not practical if the user has no knowledge of the structure at all.
   One important difference is thus that keyword querying aims at accommodating the lack of knowledge about the structure in a more fundamental and comprehensive way and thus has a philosophy and caters to a community that are different from those of traditional query languages.

A second, complementary characteristic of keyword query languages is how they are realized:

1) The majority of keyword query languages are implemented as a stand-alone system that handles the query evaluation and determining and, where applicable, ranking of the return entities.
2) Another group of keyword query languages translate the keyword queries into another query language and thus outsource the query evaluation.
3) Keyword-enhanced query languages combine conventional query languages like XPath or XML-QL with keyword-querying techniques and thus build on existing systems.

B. Using structural information for keyword querying

As mentioned, keyword querying is an established technique in various Web applications like search engines.

In these applications, most queries are keyword-only queries although for instance Google supports a limited set of label-keyword-like constructs in queries like `allintitle:XML` which retrieves Web sites that have the word *XML* in their *title* element.

But overall, querying the structure of Web documents is only enabled to a very limited degree, for example, it is not possible to specify an arbitrary HTML tag as a surrounding context for a keyword. At least in the case of Web search engines which process an extremely big amount of data (in July 2008, Google stated their link processing system had found more than 1 trillion individual links, although not all pages found are indexed[14]), this can be attributed to the fact that indexing and retrieving structural information would increase the data and processing load even further, decreasing the efficiency of search.

Amazon on the other hand which operates on less numerous, and homogenous data in a limited domain, offers advanced search[15] (see Figure 4) for various categories of products like books and magazines. Here, the user can fill in values for a number of given attributes, for instance author and language when she is looking for a book. While the advanced search is realized as filling out a form, it

[14] http://googleblog.blogspot.com/2008/07/we-knew-Web-was-big.html
[15] http://www.amazon.com/gp/browse.html?node=241582011

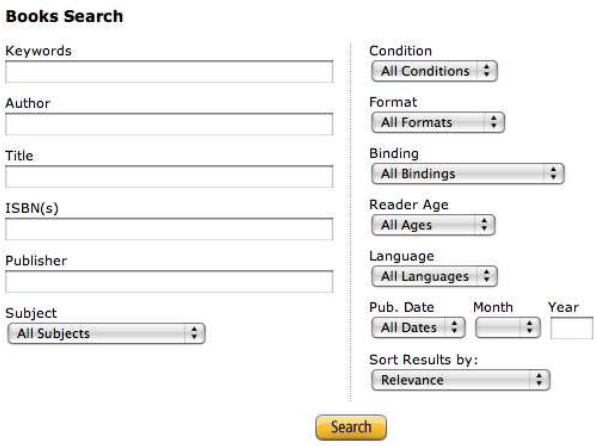

Fig. 4. The Amazon advanced search interface

essentially constitutes the equivalent of a very limited label-keyword query language. For example, providing *"XML"* as a value for the *"title"* field in the form can be seen as being equal to a query *"title:XML"*.

However, this form of label-keyword querying relies on clean, structurally homogenous data and thus does not constitute a suitable solution for flexible and versatile querying of generic XML or RDF data.

1) Computing query answers: In keyword querying on the Web, some structural information may be taken into account when ranking the results, for example by assigning different scores depending on whether a keyword occurs in the title or is printed in big or bold text [36], but the structure of a document plays no role when determining the return value type which is fixed. Apart from efficiency, two reasons can be seen for this. First, Web or Wiki pages are typically of a comparably small size, and it is thus reasonable to return results at the granularity of whole Web pages. Secondly, in the case of domain-specific querying on a limited, homogeneous data set like on shopping Web sites, querying only serves one task, i.e. finding matching products and there are only few types of objects, e.g. books and DVDs, and the return types (or the information initially displayed in the results list) can easily be predefined. For example, keywords matching a book might yield a return entity of type book which by default displays the title, author and price, while the return entity for DVDs might show the title, price and region code.

On the other hand, when applying the concept of keyword-search to RDF or XML documents, we may be dealing with a single big document that e.g. represents a bibliography or an address book which contain thousands of entries or more. In this case, it is not meaningful to return the whole document, but only parts of it. Defining return types manually may be feasible but is harder to realize than in the above-mentioned examples since the data may represent many different types of entities.

Therefore, it is preferable to employ some method to automatically determine a useful return value, a semantic entity. In order to achieve this, matched nodes have to be connected, that is, grouped according to their membership in semantic entities, particularly when there are several matches for some or all keywords. Note that grouping into semantic entities also serves the goal of establishing the domain of the answer, potentially enabling the targeted selection of return values like retrieving the publication years of all books by a certain author.

Additionally, a metric for ranking these result elements that, unlike rankings of HTML documents, performs at the granularity of the returned structures is needed. That means that the entities to be ranked are not complete documents but fragments of structured data. Finally, RDF and often XML have a deeper, more semantically meaningful structure than HTML, meaning that they profit more from a more comprehensive exploitation of the structure of the results in ranking.

However, note that XHTML is an application of XML which illustrates that it is necessary to keep in mind that XML documents are not always data-centric like bibliographies (Figure 1) but may also be document-centric and represent a text and its structure or formatting (as shown in Figure 5). A truly versatile keyword-based query language for XML should ideally yield useful results for both kinds of documents (and any that might exist in between) and not impose restrictions on the type of documents.

As an illustration of the return value problem, consider the XML document representing a bibliography in Figure 1. A query $K = \{w_1, ..., w_k\}$ matched on an XML data set T yields the result $L = \{L_1, ..., L_k\}$ where each $L_i = \{v_1, v_2, ...\}$ consists of all nodes v which contain w_i.

Applied to the data in Figure 1, the first word in the query $K = \{Smith, Web\}$ (conjunction is assumed here and in the following) has one match, the content of node 11 which is the last name of one of the authors of an article ($L_1 = \{11\}$), while "Web" matches the titles of both articles and thus $L_2 = \{3, 17\}$.

Returning only the matched nodes would not provide useful information for the user, as would returning the whole document which might contain many more entries. Given the nature of the data and the query, the user can be assumed to be interested in receiving information about articles whose data contain his search terms. That means that the "article" nodes in the data govern the subtrees which constitute meaningful semantic entities that should be returned, as a whole or in part, as answers to the query.

A common approach to the evaluation of matches in keyword-based query languages is to find the most specific element that is an ancestor to at least one match instance of each keyword and to construct a return value from this node, e.g. take the subtree of which it is the root. The idea behind this is that the ancestor-descendant relationship indicates a strong semantic connection, especially when

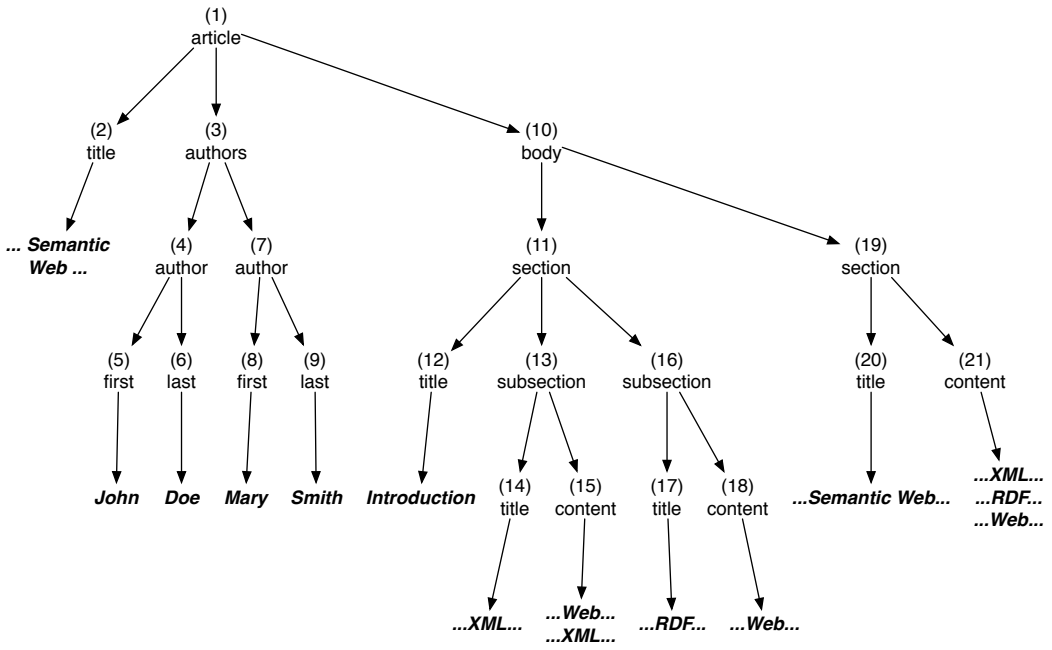

Fig. 5. Document-centric XML representing an excerpt of an article

the distance between ancestor and descendant is small. Correspondingly, a node which is the closest common ancestor of instances of all keywords is assumed to encode the most specific concept that the keyword matches have in common.

In the case of tree-shaped XML data, this is defined as the Lowest Common Ancestor (LCA) [108], a common concept in graph theory which takes an answer set S_i and computes the lowest node that has all nodes in S_i among its descendants. Depending on the application and specific algorithm, $S = \{S_1, S_2, ...\}$ may contain either all answer sets that cover the query exactly, i.e. contain one matched node for each keyword ($\|S_i\| = \|K\|$), or allow the answer sets to contain more than one match instance for each keyword.

In this example, assuming the latter case, the three answer sets are $S_1 = \{11,3\}$ $S_2 = \{11,17\}$ and $S_3 = \{11,3,17\}$ with the respective LCA nodes $LCA(S_1) = 2$, $LCA(S_2) = 1$ and $LCA(S_3) = 1$. Intuitively, the subtree governed by node 2 constitutes the domain of the best answer of the query and is preferred to the other answers which are not specific enough and should not be displayed or should be ranked lower.

Several adaptations of LCA have been proposed to remedy the problems of false positives, that is, results which are not relevant. These variants, which will be presented in the following, reduce the set of answers by filtering out false positives. As will be shown, in this process, false negatives can be introduced, that is, not all relevant answer are retrieved.

But false positives and false negatives are not the only problem that LCA grouping and related methods have; sometimes, a relevant answer may not be contained in the LCA subtree. For example, a query $K = \{author : Doe, author : Smith\}$ yields node 5 as the LCA, implying an answer that is not necessarily informative since it is only concerned with the authors' names, while it can plausibly be assumed that the user is interested in further information like the titles of the articles they coauthored..

Basic approaches to ranking use the length of the paths from the matched nodes to the LCA node to determine ranking order, shorter paths are assumed to mean that the answer is more specific, that is, better.

For both tasks, determining the domain of the return value and ranking, the structure of the queried data set proves helpful in finding a solution and thus structural analysis is employed in keyword-based query languages even though the structure may not always be queried explicitly.

The majority of research is concerned with querying XML data, but keyword-based query languages for RDF data, more complex to realize because of the graph structure, labeled edges and blank nodes, do exist and will be presented in the following. Familiarity with XML, RDF and their data models is assumed, see Section II.

C. Keyword Query Languages Implemented as Stand-alone Systems

In the following section, the approaches are presented grouped according to their type of technical realization as outlined above as well as the kind of data they operate and are sorted in chronological order.

1) Querying XML:

a) Xkeyword: Unlike almost all later approaches to XML keyword querying which can be applied only to tree-shaped data, XKeyword[16] [15], [111] can be used on XML graphs and does not require the XML data to have one common root node. As in most other keyword query languages, queries are simply keywords whose list is assumed to be in conjunction. Keywords are matched both on node labels and values. To achieve semantically meaningful return values, the XML schema graph is manually grouped into possible return types, *target objects*, which are annotated with their relationships to other target objects. For example, a target object of type *article* could consist of article, author and title nodes and stand in a *contained in* relation to a target object of type *proceedings*.
The system stores the XML data in a relational database as a set of connection relations and an inverted index that indicates for every keyword the elements in which it occurs. Queries are then processed by retrieving the relevant objects for the keywords and generating minimal cycle-free subgraphs that contain all input keywords. These in turn can be mapped onto subtrees of the target object graph, yielding the query results. The results are generated in parallel, meaning that smaller results are generated first, resulting in a ranking of the results according to size.

The results can be be displayed in a list or as a *presentation graph* that can be navigated through clicking and expanding results. Trivial and duplicate matches are initially hidden and results are grouped by their schema.

b) XSearch: [64] is a label-keyword query language for XML that includes a ranking mechanism. Search terms can be of the form $l:$, $:k$ or $l:k$. A term $l:k$ matches a node if the node has label l and a descendant in whose content k is contained. All terms are optional unless prepended with *"+"*.

Matched nodes are grouped into entities according to the *interconnection* relationship which says that the path from two nodes v_1 and v_2 to their LCA may not contain distinct nodes with the same labels except for v_1 and v_2. An answer set contains only one match for each keyword in the query and is interconnected if either it contains a node, the *star center*, that is interconnected with all other nodes in the set (*star related*) or if all nodes are pairwise connected (*all-pairs related*). The type of connection condition may be chosen depending on the data.

A query result is defined as an answer set that fulfills the selected interconnection constraint.

As an example, consider the query $K = \{+last:Smith, +:Web\}$ evaluated on the example data. As in the similar query before, $L_1 = \{11\}$ and $L_2 = \{3,17\}$. Since every answer set must have the cardinality of the query and every keyword must be matched, the answer sets are $S_1 = \{11,17\}$ and $S_2 = \{11,3\}$. The shortest path between objects 11 and 3 contains every node label only once which means that they are interconnected. On the other hand, nodes 17 and 3 are not interconnected since nodes 2 and 16 which both lie on the path between the respective nodes and the LCA node "bib" have the same label, "article". The only answer to the query is thus $S_1 = \{11,3\}$. The interconnection relation in this case avoids grouping matches together that belong to different articles as simple LCA-based grouping does.

Since there are only two elements in each S_i in the previous example, the interconnected nodes are both star-related and all-pairs related. However, since star-relatedness is a relaxation of the constraint of all-pair relatedness in the sense that for a set of nodes to be all-pairs related, every node has to be a star center, this is not always the case as illustrated by the following example: The query $K = \{+last:Smith, +last:Doe, +year:2005\}$ yields the answer set $S_1 = \{11,8,4\}$. Nodes 11 and 8 are not interconnected since the path between them passes two nodes with label "author". Node 4 however is interconnected with both 11 and 8. Consequently, $S_1 = \{11,8,4\}$ is not a query answer if all-pairs interconnection is used, but it is according to star related interconnection.

Grouping using all-pairs related interconnection is thus more restrictive than star related interconnection which in turn is more restrictive than simple lowest common ancestor calculation.

The above example also illustrates that all-pairs interconnection can lead to false negatives, since S_1 is a valid answer to the query K. Additionally, both types of interconnection semantics are sensitive to false positives when node labels are different but refer to similar concepts. In Figure 23, the query $K = \{+:Smith, +:2003\}$ with $S_1 = \{11,18\}$ and $LCA(S_1) = 1$, the root node is wrongly returned as a result of the query since "article" and "book" are different labels but signify conceptually related entities.

The query answers are ranked using the vector space model [171] and the tf-idf measure [115] applied at the granularity of individual nodes. Other factors in calculating the ranking score distance between the nodes in the answer set and the number of pairs of nodes that stand in an ancestor-descendant relationship, since this relationship generally indicates a strong connection.

The system is implemented using inverted indices for keywords and labels, an interconnection index and a path index. The interconnection index contains the pre-computed interconnection relations between all pairs in a document, while the path index stores for each keyword the paths by which it can be reached which allows to compute answers in ranking order when only the subtree size and the number of ancestor-descendant pairs are considered. The interconnection index which stores the interconnection relationships between nodes can be pre-computed or generated incrementally online as queries are evaluated.

c) XRank: [106], as the name suggests, puts a bigger focus on result ranking and employs more refined ranking techniques than the above-mentioned approaches.

[16]Project Pages: http://db.ucsd.edu/XKeyword/

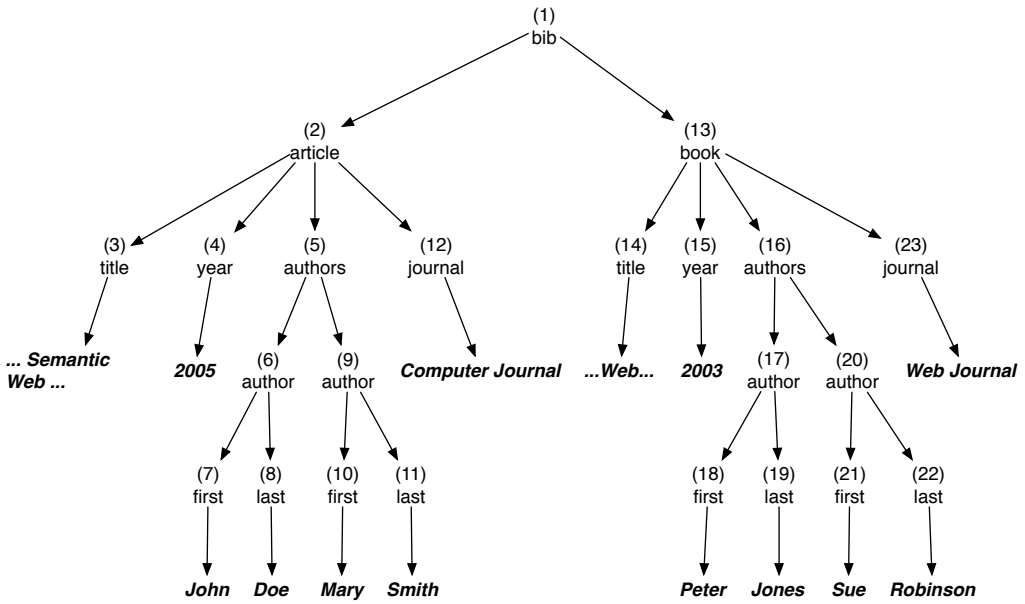

Fig. 6. False positives in interconnection semantics

The system allows for querying a mix of (graph-shaped, i.e. containing hyperlinks) XML and HTML documents. When dealing with XML, query results are XML fragments, but XRank behaves like a traditional Web search engine when dealing with HTML documents, returning complete documents as search results. XRank is one of the few query languages presented here that assume document-centric XML and exemplify the grouping on such data.

XML query evaluation in XRank proceeds by first matching the keywords on the content of nodes. Hyperlinks are ignored when calculating query results. A query result then is computed by finding R_0 the set of nodes that contain at least of instance of each keyword in the query via an ancestor-descendant relationship.

A query result node then is a node in R_0 for which it holds that, for each keyword, it contains at least one match instance that is not contained in any of its descendant nodes also in R_0. Formulated in terms of the Lowest Common Ancestor, the procedure yields those LCA nodes which either are not ancestors to any further possible LCA nodes or, if they are, which are also LCA nodes when ignoring the keyword matches in the contained LCA subtree.

As an example, consider the query $K = \{XML, Web\}$ evaluated on the data in 7. The keyword match lists are $L_1 = \{13, 15\}$ and $L_1 = \{2, 12, 13, 16, 19\}$. Based on this, some exemplary answer sets are $S_1 = \{13\}$, $S_2 = \{15, 16\}$, $S_3 = \{13, 12\}$ and $S_4 = \{15, 19\}$. S_1 consists only of one node which contains all keywords, meaning that the LCA of S_1 is identical with its element, node 13. Since this node is the LCA node and does not have any children, it is a result of the query. Similarly, node 14, the LCA of S_2, is also a result node. On the other hand, S_3 and S_4 both have the LCA node 11 which is an ancestor of nodes 13 and 14, that, as explained, are themselves LCA nodes. $k_2 = Web$ has an occurrence contained by 11 which is not part of an LCA, namely in node 12. However, there is no match of $k_1 = XML$ in a descendant of node 11 which is not also contained in an LCA. Therefore, node 11 cannot be a return node.

The intent behind ruling out non-minimal LCAs is to retrieve only maximally specific query results. However, since document-centric XML represents a cohesive text where structurally close elements can be expected to be strongly interrelated in their topic, it is also of interest to account for such matches.

The XRank grouping mechanism is susceptible to the same types of false positives are LCA and interconnection semantics (in the case of synonym or near-synonym labels) are, that is, unrelated entities may be grouped together.

Three criteria, specificity, keyword proximity and the connections between elements are used to rank the results. *Specificity* refers to the distance between the matched leaf nodes and the return node, while *keyword proximity* means the distance between the keyword matches themselves. Specificity –vertical distance– and keyword proximity –horizontal distance– thus combine into a two-dimensional proximity metric. Finally, a variant of Google's PageRank, ElemRank, is used to let the links between elements factor into a result node's ranking value.

The PageRank value of a Web site represents the probability of reaching it through randomly following links [36]; the algorithm employs link-based propagation of ranking values. ElemRank is an adaptation of PageRank that takes

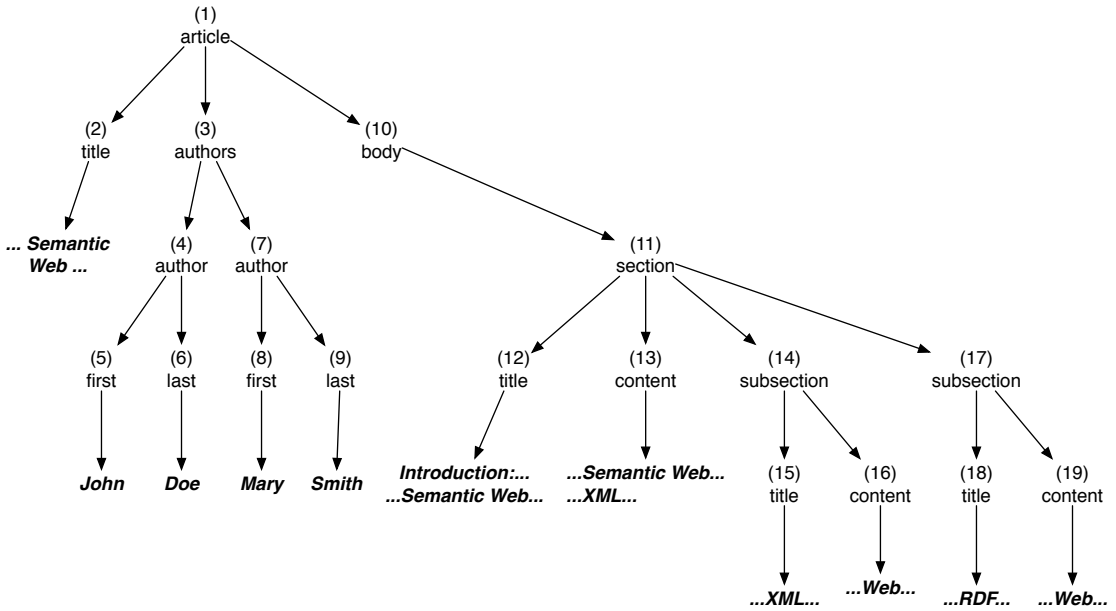

Fig. 7. Document-centric XML data

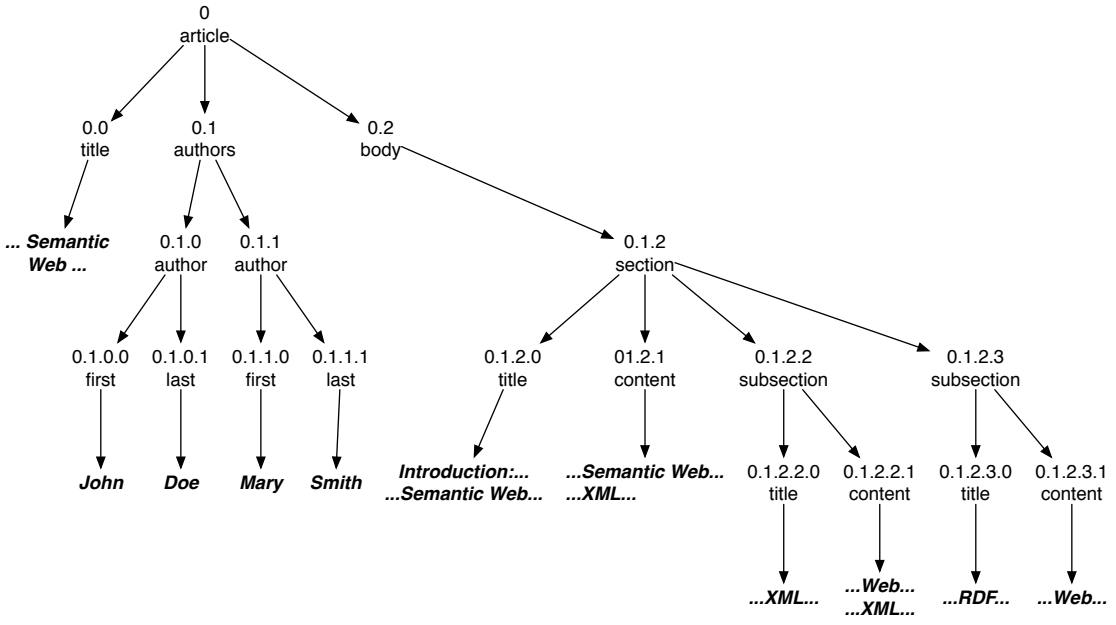

Fig. 8. XML data from Figure 7, enumerated with Dewey IDs

specific characteristics of XML data into account, namely the bi-directional propagation of ElemRanks through links, the aggregation semantics for reverse containment relationships and the distinction between containment links and hyperlinks. While hyperlinks are ignored when matching the keywords, they are considered when calculating ElemRanks. Since containment links, the parent-child relationship between XML elements, represent a stronger relation than hyperlinking through e.g. IDREFs, the two are handled separately with the propagation of ElemRank value between elements connected by containment edges being bi-directional. Additionally, the ElemRank of a node is defined as the sum of the ElemRanks of its children, meaning that the ranking values of an entity's subparts in turn combine into that entity's ranking value.

The ranking value of each instance of a keyword match is then calculated as its ElemRank value, scaled by a decay factor that is inversely proportional to the distance between the result node and the keyword match.

Finally, the ranking value of the result tree is the sum of the ranking values of the contained keyword occurrences multiplied by a measure of keyword proximity which is based on the size of the smallest text window containing all matches.

If a keyword has several occurrences in the subtree governed by the result node, the value of the node with the highest ElemRank value is used.

In summary, the criterion of specificity is realized as the decay scaling factor where the decay increases as the distance between a keyword occurrence and the result node grows, meaning that the ElemRank calculated from the link connections between the elements becomes smaller. The keyword proximity criterion on the other hand is represented as the scaling factor of the overall ranking value of the result, here, a bigger distance between the keyword occurrences corresponds to a lower scaling factor.

The calculation of the result nodes themselves is implemented via evaluation of Hybrid Dewey Inverted Lists. Dewey ID enumeration [183] is a system to enumerate nodes in an XML tree. A Dewey ID is a vector that summarizes the path from the root node of a document to a node. Figure 8 shows the data from the previous figure enumerated using Dewey IDs. In this enumeration scheme, the ID of an ancestor node is a prefix of the IDs of its descendants. The Lowest Common Ancestor of a set of nodes can thus be easily computed by determining the longest prefix shared by all nodes' Dewey IDs. For example, $S_2 = \{15, 16\}$ and $S_4 = \{15, 19\}$, using Dewey enumeration, become $S_2 = \{0.1.2.2.0, 0.1.2.2.1\}$ and $S_4 = \{0.1.2.2.0, 0.1.2.3.1\}$ and this information suffices to compute the respective LCA nodes, 0.1.2.2 and 0.1.2. This property allows for the efficient computation of result nodes in a single pass when all keywords are stored in an inverted list associated with their Dewey ID.

The authors present two techniques for the calculation of result nodes which can produce the top-k results, meaning

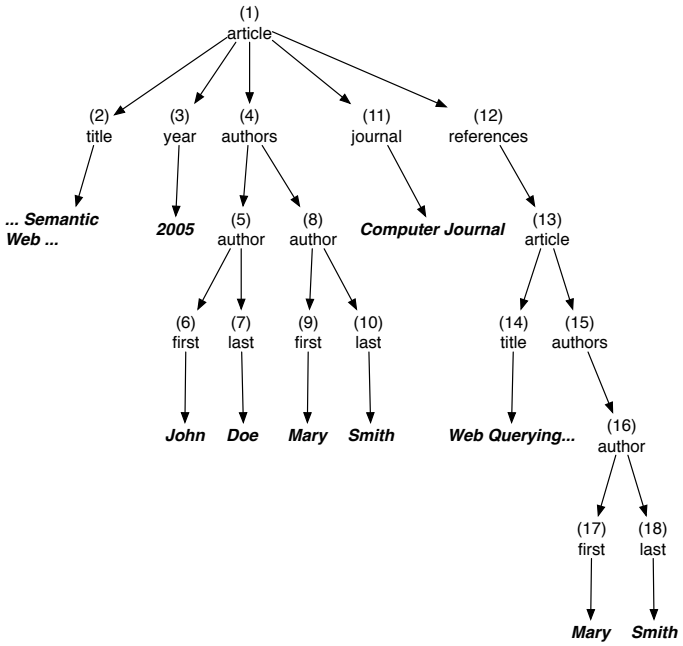

Fig. 9. Sample XML data

that not all results have to be generated before ranking can occur. A hybrid model allows for the combination of both techniques depending on whether there is a high or low correlation between the occurrences of the keywords.

d) XKSearch: [17] [196] introduces the notion of the Smallest Lowest Common Ancestor which extends the definition of LCA by a minimality criterion. Only LCA nodes that do not have further LCA nodes among their descendants are also SLCA nodes. Note that this definition is stricter than that of result nodes in XRank in that SLCA generally forbids LCA nodes that have LCA nodes among their descendants, while XRank only constrains the context in which an LCA may contain another LCA. Further, SLCA only allows one match instance for each keyword in an answer set.

SLCA addresses the problem of false positives as described in Section VI: Evaluation of the query $K = \{Smith, Web\}$ on the data in Figure 1 leads to the resulting LCA nodes 2 and 1. Node 2 represents a node of type *article* and constitutes a meaningful result, while node 1 is the root node of the document and the keyword matches are distributed over two different articles.

According to SLCA semantics, only node 2 is a suitable result node since it does not contain LCA nodes. Node 1 is a LCA node but not a return node since it is an ancestor of another LCA node, node 2.

However, false positives are still possible when SLCA is used for grouping. The query $K = \{Smith, 2003\}$ has $S_1 = \{11, 14\}$ and $LCA(S_1) = 1$ as a result according to SLCA. This

[17] Project Pages: http://db.ucsd.edu/People/yu/xksearch/index.htm

query answer is not meaningful since the keyword matches are distributed over two different articles, but is not filtered out since there is no valid result and thus no further LCA in the data.

Additionally, disallowing nested LCAs can also lead to false negatives, for example when the same query, $K = \{Smith, Web\}$, is applied to the data in Figure 9, resulting in, among others, the answer sets $S_1 = \{18, 14\}$ and $S_2 = \{10, 2\}$ and the corresponding groupings $LCA(S_1) = 13$ and $LCA(S_2) = 1$. Both LCA nodes represent articles which contain both of the query terms and thus can be considered suitable results. However, since the second LCA is an ancestor of the first, SLCA filters out the latter. Consequently, only the article referenced in the other article is retrieved as a result.

The authors present two efficient algorithms for calculating SCLA nodes, Indexed Lookup Eager and Scan Eager, that are based on the observation that a set of nodes lying close together in the tree translates to their LCA being in closer proximity to them, meaning that for each keyword match, only the nearest two matches for the other keywords have to be taken into consideration when computing the SCLA.

e) Pradhan: [164] presents a keyword-only approach targeted at querying XML data representing textual documents, e.g. Web sites or books.

Document-centric XML has a more variable schema and weaker semantics than data-centric XML since the XML structure may represent formatting or structural subdivision in e.g. paragraphs.

Due to these differences, Pradhan argues that while the LCA subtree containing one match for each keyword may be a suitable query answer for data-centric XML, it does not constitute a meaningful query answer in the case of document-centric XML documents where adjacent paragraphs may each contain one or more of the query terms.

A query is matched on node content and a query answer may contain more than one match instance of each keyword. An answer is constructed from sets of answer fragments where each set contains a matching node for one keyword. These fragments are then joined, connected along the shortest path through the XML tree, to yield answer candidates. All combinations of elements from the sets are computed and filtered, for example by the size or height of the resulting answer. The query answer may be a single node if all keywords are matched in its textual content.

Since the computational cost of this procedure can be very high when there are many keyword matches, the author suggests optimizing the calculation by iteratively applying joins and removing redundant elements in the fragment sets, that is, fragments that would be subsumed by a join of some other fragments in the set, or, more informally, fragments that lie in the path that directly connects two other fragments in the same set.

In addition to filtering, the number or query answers is reduced by eliminating duplicates.

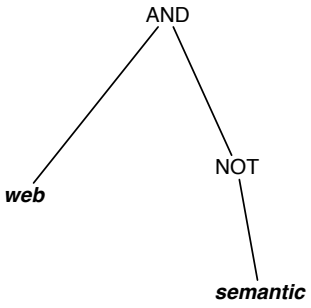

Fig. 10. Query Tree

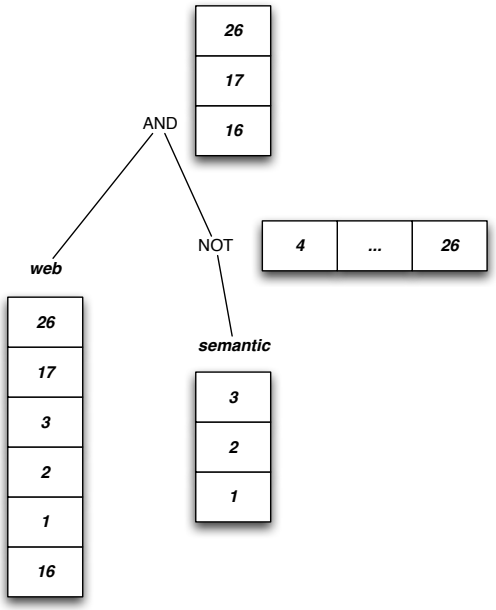

Fig. 11. Query evaluation

f) Abbaci et al.: [2] present a keyword-only query language that, compared to the other approaches discussed here, offers a relatively complex syntax, namely the operators AND (conjunction), OR (inclusive disjunction), INC (inclusion, one operand must occur in a node that is a descendant of the node containing the other operand) , SIB (sibling) and NOT (negation) and additionally parentheses to indicate precedence. The query language operates on XML documents and keywords are matched on node labels as well as node content.

Query evaluation is realized by transforming the query into a binary tree where the leaf nodes contain the keywords and the other nodes contain the operators. Next, sets of matching elements are constructed for each leaf node, that is, each keyword. If a node v_i contains a keyword w_i, all

ancestors of v_i are also represented in the list of matches for w_i since they contain it indirectly. The data structure in which the matches are recorded represents the ID of the node in which the keyword term occurs, the type of the occurrence and the distance from the keyword match to the respective node (i.e. the distance is 0 when the node contains the keyword directly). The answer sets for each leaf node are then further processed by applying the operator specified in a node to the answer sets of its children. AND corresponds to the intersection of two sets, OR to the union and NOT to the difference. INC and SIB are realized via constraints on the distance from the keyword match. The query tree is processed in a bottom-up fashion and once the root node of the query tree has been processed, only the nodes matching the full query remain. Since indirect matches via ancestors are included in the answer sets, in the case of conjunctive queries, the LCA node is among the query results.

As an illustration, consider the query "Web AND NOT semantic" evaluated on the data in Figure 1. The query tree of this query is displayed in Figure 10. The query is then evaluated by first adding information about matching nodes to the leaf nodes as shown in Figure 11. Note that for simplicity's sake here only the IDs of the matching nodes are shown in the lists and not the information about the match types and distances which is only used for ranking. The keyword "Web" is contained directly in nodes 26, 17 and 3 and indirectly, i.e. via a descendant, in nodes 2, 1 and 16. "semantic" is contained directly in node 3 and indirectly in all ancestors of 3, that is, nodes 2 and 1. To apply the NOT operator, the difference between the set of all nodes in the XML data and the nodes containing "semantic" is taken, yielding the set of nodes that do not contain "semantic" either directly or via an ancestor. Finally, to find the nodes that fulfill both conditions, the intersection of the sets of nodes that contain "Web" and those that do not contain "semantic" is taken. The node list after application of the AND operator is the final result since the root node has been reached.

g) Saito et al.: The label-keyword query language for XML data conceived by Saito et al. [170] has a syntax that allows for the use of some XPath operators and constructs, for example paths or parts thereof can be specified through the use of the child and descendant operators. The query language might thus also be considered a keyword-enhanced query language, but since the extent of XPath and its expressions that can be used in this query language are unclear and not discussed further, we consider the query language to be a keyword-label query language that offers some support for paths rather than the other way around. An example of a label-keyword query term is $k = //last/text() =$"$Smith$", while only the nodes containing "Smith", regardless of their labels, are matched for $k =$"$Smith$".

The method used for grouping matched nodes is called Amoeba join. An Amoeba is an answer set which contains its LCA node, that is, one of the nodes in the answer set is in an ancestor-descendant relationship with all other nodes in the set. The authors refer to this as the nodes being bound to the Amoeba root and state that the relationship between nodes in an answer set is "very weak" if their LCA node is not also an Amoeba root.

For example, the query $K_1 =$ {"$Smith$","Web","$article$"} applied to the data in Figure 1 yields, among others, the answer sets $S_1 = \{11, 17, 2\}$ and $S_2 = \{11, 3, 2\}$. The former is not an Amoeba since $LCA(S_1)$=1, i.e. the root node is the LCA node of S_1. Consequently, S_1 does not constitute a result to the query according to Amoeba join. S_2 on the other hand has node 2 as its LCA. Since node 2 is also contained in the set, S_2 is an answer to the query.

However, Amoeba join can be too restrictive, leading to false negatives and unintuitive behavior; the query $K_2 =$ {"$Smith$","Web"} finds no results in the data in Figure 1 although it is a relaxation of K_1 and all query answers of K_1 should also be answers to K_2.

On the other hand, recursive elements can lead to false positives as discussed in [186].

h) Li et al.: The work of Li et al. [132] is concerned with improving on the shortcomings of LCA and SLCA when grouping the matches and determining the return entity. As discussed above, the application of LCA and SLCA can lead to matches that are not very informative for instance when a user queries for a name and words from a paper's title and is returned two distinct publications, one written by an author with the name given in the query and one having the other keyword in its title.

On the other hand, nested XML structures can lead to false negatives, that is, results that are not retrieved although they would be an informative answer to a query. As explained in Section VI, when the XML representation of an article matches all keywords and contains another article that also matches the query in its references, only the latter will be returned according to SLCA semantics.

To resolve these problems, the authors introduce the concept of Valuable LCA (VLCA). A VLCA is an LCA where the keyword-matching nodes are homogeneous. A set of matched nodes is defined to be homogeneous if no node label in the paths between them (excluding the labels of the matched nodes themselves) and their LCA occurs more than once. That means, each element in the set of the labels encountered when traversing from each matched node to the common LCA should be unique. For example, in Figure 1, nodes 7 and 22 are not homogenous since there are two nodes with label "article" in the path between them, nodes 2 and 16. Nodes 3 and 5 on the other hand are homogenous.

VLCA is conceptually identical to all-pairs related interconnection semantics in XSearch and has the same problems with false positives and false negatives as described above in VI-C1b.

To achieve faster computation of VLCA nodes, the authors present the notion of Compact VLCAs and Compact answers. Compact VLCAs are compact in that they enforce

maximally specific results.

A Compact LCA node is the LCA node of an answer set that dominates all the nodes in the set. A node v_i dominates another keyword-match node v_j if there is no answer set involving v_j that has an LCA which is a descendant of v_i. Put more simply, an LCA is only a Compact LCA if it holds for all contained matched keywords that they could not be part of a grouping of matches that has a more specific LCA.

Accordingly, a CVLCA is a CLCA that is also a VLCA. The Compact Answer to a keyword query contains only the CVLCA node and the labels and content of the matched nodes governed by it.

The authors further present a stack-based algorithm to efficiently calculate Compact Answers that exploits the fact that one matching node can only have one CVLCA.

i) XSeek: [18] [134], [135] places emphasis on methods of inferring return structures from keyword queries to XML data. Most approaches to keyword querying XML focus on grouping the matches into semantic entities and establishing their root nodes. The return value is then taken to be either only the root node, the whole semantic entity (that is, subtree) or the paths from the keyword matching nodes to the root node.

The authors point out that all these approaches are suboptimal since the second strategy may lead to large return trees of which only a small portion is relevant, while the first and the third do not provide enough information to be helpful.

In addition to these points raised by the authors, more targeted return values also allow for a more sophisticated and controlled querying that has some more of the power of traditional query languages. As mentioned, a query for two authors' names on bibliographic data can reasonably be expected to return information about publications they co-authored, yet none of the approaches presented so far that use LCA-based grouping would yield this result.

XSeek matches queries both on node labels as well as on content and employs VLCA to group the resulting matches. The keywords in the query are automatically grouped into those that express search predicates and those that specify return information. If a keyword w_i matches a node label and no other keyword in the query matches the node content of a descendant of w_i, then w_i is considered to be a return node. All nodes that cannot be determined to be return nodes are predicates.

If no return nodes can be inferred, the entities in the paths from the matched nodes to the VLCA node as well as the VLCA node's lowest ancestor entity are considered to be the return nodes. A node is considered an *entity* if it is in a many-to-one relationship with its parent. For example, a bibliography often has several article nodes among its children, making article nodes entities. These relationships can be inferred from the relations in the data or, if present, from the schema.

[18]Project Pages: http://xseek.asu.edu/intro/Home.htm

A node that is not an entity and has only one child which is a value on the other hand is considered to be an *attribute*, while nodes that are neither attributes nor entities are *connection nodes*.

According to these rules, the "article" and "author" nodes in Figure 1 are entities, "title", "year", "journal", "first" and "last" are attributes and "authors" is a connection node.

The result of a query is constituted of two parts, the return nodes and their associated information and the paths from the VLCA to the matched nodes.

The information that is displayed for each return node depends on its type, attributes are displayed as their name and value, while for entities and connections, the subtree rooted by the node is included. Since the resulting subtree may be large, child entities are initially shown collapsed.

j) Kong et al.: To allow for more flexible and complete query results, Kong et al. [128] introduce the idea of Relaxed Tightest Fragments (RTF) which is another method for connecting node matches for keyword-only queries on XML data. In addition, they present a ranking mechanism for RT fragments and an efficient algorithm for their computation.

In many previous approaches to the problem of connecting keyword matches, only one match instance of each keyword is considered for the computation of the common root; RTF on the other hand allows for multiple matches of a keyword to be present in one result fragment. The keywords in the query are matched against node contents only.

RTF imposes the constraints that, for a given answer set S_i, no subset which is also an answer set may have an LCA that is different from the LCA of S_i. Additionally, the set of keyword matches has to be the maximum set of matches for the given LCA. That is, it should not be possible to add further keyword matches to the set without the addition resulting in a different LCA. Finally, the third constraint says that no keyword match node in the set can be part of a keyword answer set whose LCA node is a descendant of the LCA of S_i.

Essentially, RTF is a variation of CVLCA where the mode of generation of the answer set and the first two constraints ensure that the result subtrees are complete with respect to the keyword matches while still being as small as needed to cover at least one instance of each keyword match.

For example, the query $K = \{XML, RDF\}$ executed on the data Figure 1 yields keyword match lists $L_1 = \{14, 15, 21\}$ and $L_2 = \{17, 21\}$ and, among others, answer sets $S_1 = \{14, 17\}$ and $S_2 = \{14, 15, 17\}$ with $LCA(S_1, S_2) = 11$. S_1 fulfills the first requirement since it contains no subset answer sets. But keyword matches could be added to S_1 without changing the LCA node, so S_1 is not a valid query answer. S_2 has an answer set subset, namely the elements of S_1, but the LCA nodes of S_1 and S_2 are identical. The only keyword match that could be added to S_2 is node 21, which would change the LCA node to node 10. There are no possible LCA nodes below the LCA node of S_2. Therefore, all constraints are fulfilled and S_2 is considered a query answer. The root node

of S_2 together with the keyword matches and the paths to them form the return entity, a Relaxed Tightest Fragment.

RTF can lead to false positives when keyword matches are distributed over several unrelated semantic entities (see above).

The RTFs for a query could be calculated by generating all candidates and applying the constraints in order to filter out results which are not RTFs. Since this would be very costly, the authors present the Layered Intersection Scan Algorithm which efficiently generates RT fragments. The algorithm is based on the observation that, given lists of matches for all keywords, the intersection of all Dewey prefixes can represent an RTF root node under certain conditions.

The ranking system XKSMetaRank computes an RTF's meaningfulness as the weight of its root node. This in turn is computed recursively as a function of the weight of a node which is given through its label's weight and the weight of its descendants. At the lowest level of the recursion, the weight of a leaf node is given as the overlap between the keyword query and the content of the leaf node.

2) Querying RDF:

a) QuizRDF: [71] is a browse and query system for Web pages that combines full text search with querying, where present, RDF annotations. The motivation for this combined querying is that not all Web data is annotated and furthermore it is not possible to capture every detail of the content of a text in its annotations, meaning that combining full text search with RDF querying can improve recall.

QuizRDF is described as an "information-seeking system" where finding information is an interactive, gradual process rather than a targeted, single-step search. This approach is similar to what [178] propose and allows users to explore the data, refining their queries as they gain more information about the nature of the data.

Initially, a so-called ontological index is created from both the textual content of a Web site and its RDF annotations which are linked to the RDF Schema [35] ontology. This index can then be queried using keyword search which returns a list of matching Web sites, ranked using the tf-idf [115] measure. For the Web sites in which RDF annotations are present, the search results can then be refined by restricting matches to a certain RDF resource class and entering literal values for RDF properties. To provide information about the ontological structure, QuizRDF also displays superclasses of the currently selected class as well as relations to to other classes.

Suggested enhancements of QuizRDF include the ranking of clusters of documents belonging to the same resource class and chaining in the form of allowing the combination of several queries at the RDF level.

b) Q2RDF: [166] is a system for querying RDF data using keyword-only queries and ranking the results that is relatively similar to Q2Semantic. Q2RDF operates on an RDF sentence graph [198], an undirected graph consisting of RDF sentences and the connections between them. An RDF sentence is the set of all RDF triples that are *b-connected*, that is, that contain the same blank node. B-connectedness is transitive and RDF statements which do not contain blank nodes are separate sentences. The label of a node in an RDF sentence graph are the words contained in the subjects, predicates and objects it summarizes. Any RDF graph can be collapsed into an RDF sentence graph. Figure 12 shows an example of an RDF graph and its grouping into sentences. Due to the transitivity of the b-connectedness relation, RDF sentences are not stable and may change when a blank node is introduced in another part of the RDF graph (compare Figures 12 and 13).

In the preprocessing step, an inverted index which indicates which word appears in which sentences and the path index are created. The path index indicates for each node which other nodes it can reach and all shortest paths between nodes can be constructed from it. The shortest paths are calculated using Dijkstras single source shortest path algorithm.

When a user poses a query, the keywords are first mapped onto the RDF sentences in which they appear.

The goal is then to find answer trees, that is, trees that contain all keywords and in which all leaf nodes contain contain at least one keyword. This is realized by starting from the matched nodes and gradually visiting nodes until a path connecting all matched nodes is found. The next node to visit is determined by first choosing the set of keyword match nodes with the lowest cardinality, i.e. the smaller number of elements and expanding them first. Then, the closest node from the node currently being expanded is visited and added to the set of nodes to expand.

Using this method, it is possible to generate the top-k answer trees without having to generate all answer trees first if only tree size is considered as a measure of goodness since the length of the paths and thus the results trees grow as the number of visited nodes increases (the same is true for finding the top-k lowest cost answer trees in Q2Semantic since all cost values are positive).

The algorithm can result in isomorphic answer trees, the duplicate answers are discarded. The generated answer trees are ranked using a variant of the term frequency measure.

D. Translation Keyword Query Languages

c) SemSearch: [130] is a search engine for Web documents augmented with RDF annotations and as an output returns a ranked list of matching HTML documents. Only the RDF data but not the documents themselves are processed during query evaluation.

The syntax of SemSearch consists of pairs of subjects and keywords, connected by a colon and the operators "and" and "or" to indicate conjunction and disjunction.

During evaluation, the keywords are matched only to *semantic entities*, that is, classes, properties and instances,

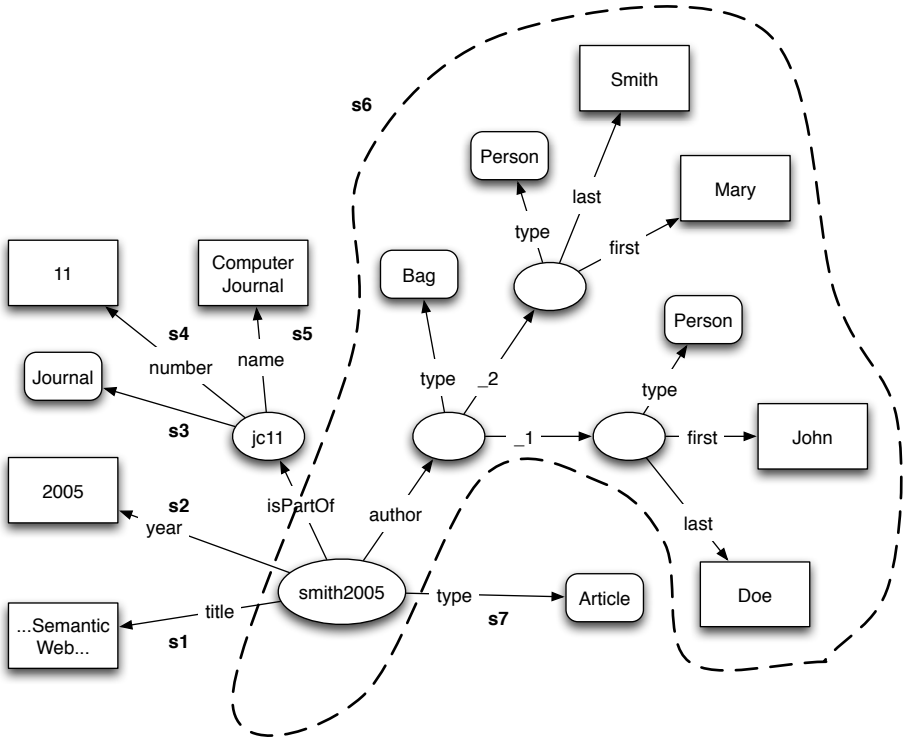

Fig. 12. RDF sentence graph

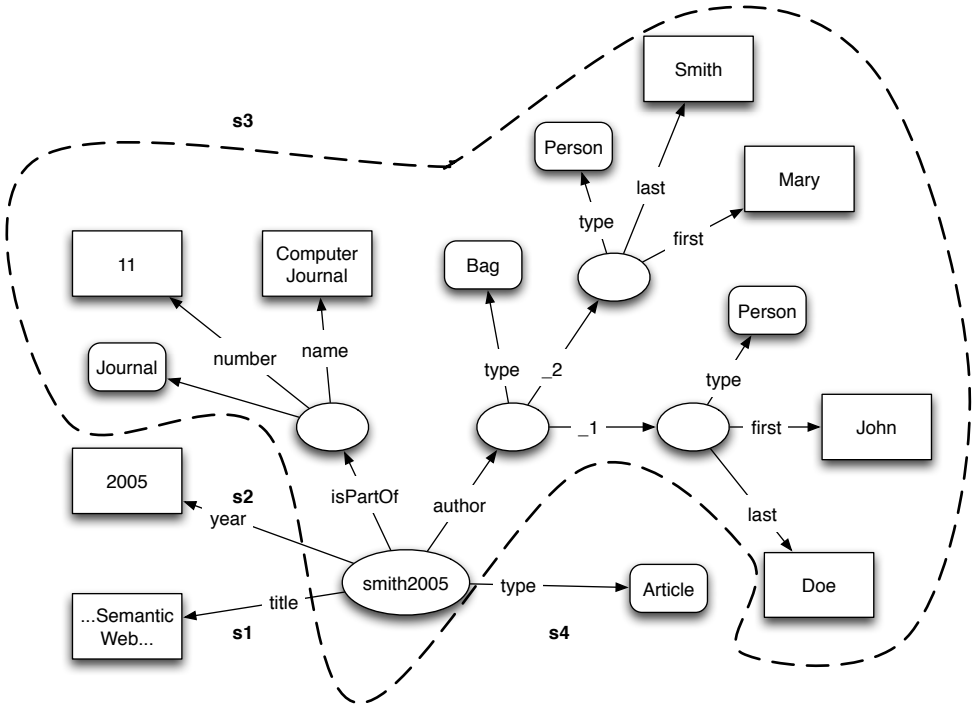

Fig. 13. RDF sentence graph

but not to relations. It is assumed that query subjects refer to RDF classes and specify the return type. If no classes match the subject, the type of the subject is determined and rules are used to infer the return type from the types of entities of the keyword and subject. For instance, Mary:John are both instances and the rule for that case says that the return type should also be an instance, e.g. an article which Mary and John co-authored.

Using the list of matching entities and their types, the user query is then translated into SeRQL through employing templates. Multiple queries are constructed if a keyword matches several semantic entities. As the number of queries can be very big when one or more keywords in the query have multiple matches, rules are employed to reduce the number of constructed queries. For example, only the most specific class is considered if there are several matches of type class. The application of the rules can be expected to decrease the recall of the search.

For ranking, two factors, namely the distance between each keyword and its matches and the number of keywords satisfied by a search result are considered; it is not clear in detail how the former is quantified or measured or how the two are combined.

Finally, the retrieved documents are displayed in ranking order, the individual results are augmented with information about the matched entities.

d) SPARK: [201] is a search system for RDF data that translates keyword-only queries into SPARQL and ranks the resulting queries. The keywords are mapped onto resources, that is, classes, instances, properties and literals, in the knowledge base. This is achieved using both the form and the semantics of the keywords. The form-based mapping uses string comparison techniques like the Edit distance [131] and in addition applies stemming [136]. The semantics-based mapping retrieves semantically related words like synonyms using thesauri. One query term can be mapped to several resources of different resource types in this process. The different mappings are augmented with confidence scores based on the similarity between the keyword and the concept.

In the next step, the query sets are constructed. If all keywords were uniquely mapped to one resource, there is only one query set, otherwise all combinations of query sets where each query set contains one resource for each keyword are generated. For each query set, a query graph is constructed using Kruskal's Minimum Spanning Tree algorithm [129] and missing relations and concepts are introduced to form a connected graph which is then translated into a SPARQL query.

Finally, the ranking scores of the generated queries are computed from the the similarity of the keywords and the concepts they are mapped to, the proportion of overlap in resources between the keyword query and the corresponding SPARQL query and the information content of the query.

e) Q2Semantic: [19] [191] provides a system for querying RDF data using keyword-only queries that are translated into formal queries which in turn can be mapped directly onto SPARQL queries. The system aims at providing higher efficiency than comparable approaches since it operates on summarized RDF graphs, RACK graphs, instead of the original data, thus reducing the data space.

Q2Semantic ranks the query results and uses Wikipedia to find related concepts for keyword query terms. These are also used to assist the user in entering his keyword query as the interface offers auto-completion for RDF literals and Wikipedia terms.

When displaying the query results, Q2Semantic also shows the portion of the RDF data used in the query as well as the translated formal query and its natural language explanation.

An RDF graph is converted into a so-called RACK graph by mapping relations, attributes, instances and attributes values onto R- and A-Edges and C- (instances) and K-nodes (attributes) respectively. R- and A-Edges and C-nodes are then clustered together if they have the same labels and, for the edges, the same connections. K-Nodes are merged when they connect to the same A-Edge and the new merged node inherits the labels of both or all K-Nodes. Costs are calculated for edges and nodes based on the number of elements merged to obtain them.

A keyword query is first matched against an inverted index which stores the K-Node labels and is augmented with terms extracted from Wikipedia, e.g. the anchor text of articles linking to an article whose title is a K-node label, to allow for a broader vocabulary in the queries. Keywords are thus only matched to RDF attribute values. If there are several matches for one term, all are returned and used in the next step.

Starting from the matched K-Nodes for all query terms and using the cost functions of the edges as a heuristic for guiding the search, a tree is then gradually built up in the graph in a round robin fashion. To avoid recursion, repeated exploration of the same node within one path is penalized through adding a high number to the cost. A formal query is found when a root that is common to at least one instance for each keyword is reached. Note that this concept is similar to the idea of the Lowest Common Ancestor.

Since there may be several possible formal queries for one keyword query, a ranking function is employed that uses the lengths of the paths in the formal query, the scores of the matched K-nodes present in the formal query and tf-idf-like measure for determining the importance of the individual query elements to calculate ranking scores.

As mentioned, Q2Semantic and Q2RDF represent similar approaches which both summarize the initial RDF graph and then construct minimal answer trees containing all matched nodes to find the top-k results which are then

[19]Project Pages: http://q2semantic.apexlab.org/

```
select  $p
from bib -e-> article $p
    $p -e-> authors -e-> authors -e-> last -e-> cdata $a
    $p -e-> title cdata $t
where $a = "Smith"
and $t like "Web"
```

Fig. 14. An example query

```
select  meet(o1,o2)
from * -e-> cdata -e-> string o1, * -e-> cdata -e-> string o2
where o1 contains "Smith"
and o2 contains "Web"
```

Fig. 15. An example query using the meet operator

ranked using a tf-idf-like measure.

The main differences between the two approaches lie in the way in which they evaluate results –Q2Semantic translates queries into complex queries while Q2RDF retrieves the results directly–, reduce the RDF graph, the element types against which keywords are matched and the cost function that guides the search for the answer trees. Additionally, since Q2Semantic merges edges and attributes only when they have the same label while Q2RDF collapses all elements that belong to the same sentence into a node, Q2Semantic's answer trees reflect a lower granularity.

E. Keyword-enhanced Query Languages

f) Schmidt et al. : [178] present one of the first approaches to keyword querying XML data. Their goal is to enable explorative querying when the schema is unknown, that is, querying is seen as an interactive process where the user can refine her query based on the result of a previous, less specific query. The keyword querying presented in this approach is implemented as part of a traditional query language where the results of the keyword querying provide information for the construction of complex queries. The query keywords here are matched only to the content of leaf nodes and not to node labels.

The query language being enhanced [177] is a variant of SQL that allows to specify paths and path variables. A query which selects publications by authors with the last name "Smith" whose title contains "Web" from data structured as those in Figure 1 is shown in Figure 14.

A query returns the *nearest concept*, that is, the LCA node that encompasses all search terms.

The authors present the *meet* operator which, given an arbitrary number of keywords, returns their LCA node. A query using the meet operator which returns results identical to the path query in the figure above is shown in Figure 15. The output of the meet operator calculated on the Monet transform [177] of an XML document, a

```
where <article><authors><author><last>$N</last></author>
              </authors>
              <title>$T</title><year> 2005 </year>
      </article> ELEMENT_AS $E IN "bib.xml",
      $N like *Smith*, $T like *Web*
construct $E
```

Fig. 16. An XML-QL query

path-centric representation of XML data, proceeding in a bottom-up fashion from the matched leaf nodes. Nodes are contracted until the meet node is found or the root node is reached. This procedure is only suited for tree-shaped XML data, meaning that XML containing IDs and IDREFs cannot be queried.

The results can be further restricted, for instance by allowing only certain types of nodes as a result or setting a maximum on the distance between keyword-matched leaf nodes.

Further, the authors suggest the use of thesauri on the keyword search terms when only few answers are returned, since the user may not be aware of the names of the node labels. Their suggested technique thus combines explorative querying with (semantic) query relaxation.

For ranking, the distance between the leaf nodes and the meet node is used as a simple heuristic as the keyword matches can be assumed to be more strongly associated if the distance between them is smaller.

g) Florescu et al.: The work presented by Florescu et al. [88] is based on extending XML-QL with a *contains* predicate. This addition makes it possible for a query language to be usable both by naive users who do not know the structure of the data or the syntactic constructs of the query language as well as experts whose queries may require more precision than keyword-based querying can offer.

The query language presented operates on a set of documents that contain XML that does not contain IDREFs.

The *contains* predicate takes four arguments, namely a signifier for an XML element that function as the root of the subtree in which the search is conducted, a keyword, an integer specifying the maximum depth at which to search and a set of boolean expressions to constrain the type of XML element, tag or attribute name and content or attribute value, in which the keyword may appear.

This provides flexibility to the user who can fully specify XML-QL queries to the degree that her knowledge of the structure of the data and of the query language allow and who can use the convenient *contains* predicate where needed.

For example, the queries in Figures 16 to 18 all express the same intention, namely retrieving articles written by

```
where <article> <authors> </authors> ELEMENT_AS $A,
                <title>$T</title>
      </article> ELEMENT_AS $E IN "bib.xml",
      contains($A,"Smith",3, any), $T like *Web*,
      contains($E,"2005",3,any)
construct $E
```

Fig. 17. The query from Figure 16 reformulated using the *contains* predicate

```
contains($E,"Smith",3, any), contains($E,"2005",3,any),
contains($E,"Web",3, any)
construct $E
```

Fig. 18. The query from Figure 16 reformulated using the *contains* predicate

Dingle in 1999 that contain the term “Web” in the title, but they display different levels of specificity. The query in Figure 16 is regular XML-QL, Figure 17 shows a query that could have been created by a user who has only a vague idea of the structure of bib.xml, and finally query 18 presupposes no knowledge of the structure besides the presence of an article element. Note that the granularity of the return value depends on how much of the query is explicitly specified, for example, in query 18, it would not be possible to retrieve the contents of the author element, since its presence is not stated in the query and thus no variable can be bound.

The query system is implemented in a relational database where for each keyword the nodes containing it are represented in an inverted index. The authors observe that this technique leads to a storage size about ten times that of the original XML document, which may be problematic if a big amount of data is present in the system.

Compared to pure XML-QL, usage of the *contains* predicate leads to a lower precision which is not surprising given the less specific queries. It is suggested that users can refine queries incrementally, using the answer from one query to formulate a further, more specific query.

h) Schema-Free XQuery: [133] aims at enabling using XQuery without full knowledge of the schema of the XML data. To this end, the MLCAS (Meaningful Lowest Common

```
for $a in mlcas doc("bib.xml")//author
    $b in mlcas doc("bib.xml")//title,
    $c in mlcas doc("bib.xml")//year
where $a/text() = "Mary"
return <result> {$b, $c} </result>
```

Fig. 19. Schema-Free XQuery

```
for $r in doc("bib.xml")//bib[1],
    $a in mlcas $r//author,
    $b in mlcas $r//author
where $a/text() = "Mary" and $a != $b
return $b
```

Fig. 20. Schema-Free XQuery

```
for $y in mlcas doc("bib.xml")//year,
    $a1 in mlcas doc("bib.xml")//author,
    $t1 in mlcas doc("bib.xml")//title,
    $t2 in
      {
       for $a in mlcas doc("bib.xml")//author,
           $t in mlcas doc("bib.xml")//title
      where $a/text() = "Mary"
      return $t
      }
where $t1 = $t2
return <result> {$y, $a1} </result>
```

Fig. 21. Schema-Free XQuery

Ancestor Structure) function is added to standard XQuery.

An example of a query in Schema-Free XQuery is displayed in Figure VI-E0h. The result of this query are the years and titles of works by the author “Mary”. Since the MLCAS keyword is present, upon evaluation, the variables $a, $b and $c and are bound to nodes with labels “author”, “title” and “year” respectively. In order to obtain a meaningful result, the constraint that all three nodes have to be part of the same MLCA structure is imposed.

The MLCA node of a set of nodes is its LCA node given that for each pair of nodes, there are no other combinations of nodes with the same label that has an LCA node which is a descendant of their LCA node. Intuitively, this means that for all keywords, the node with the keyword label that is most closely related to the other matched nodes is found. This technique is based on the assumption that a lower LCA means a stronger connection. The concept (and the problems) of MLCA is very similar to that of SLCA with the difference that SLCA does not impose constraints on the node labels. The MLCAS then is the structure consisting of the nodes among which the MLCA relationship holds.

Having determined the MLCAS, the variables are then bound to the content of the children of the respective nodes in the MLCAS. The keyword-aspect of Schema Free XQuery thus pertains to node labels and not, as in many other keyword query languages, to the content. As can be seen in the example in Figure VI-E0h, restrictions on the content are imposed in the where clause of the query.

The user can create more complex queries using regular XQuery as is shown in the query in Figure VI-E0h which retrieves co-authors of Mary. All uses of the MLCAs keyword within the same query refer to the same mlcas, but nesting allows for separate MLCA structures as demonstrated in the

query in Figure VI-E0h.

Another feature of Schema-Free XQuery that furthers the goal of creating a query language that does not require knowledge of the schema is term expansion. Term expansion means that a user does not have to indicate the exact node label in a query if she is not sure. It has been found that less than 20% of people chose the same name for a common object [93] and it can be assumed that node labels referring to the same entities are similarly varied. The solution suggested in Schema-Free XQuery is the use of an *expand* function that the user can employ to indicate that she is not sure she used the correct node label in the query. These node labels are then matched in a thesaurus which retrieves possible alternatives. To avoid having to evaluate multiple queries, this synonym relationship is represented in the database.

For the computation of MLCA structures, the authors present a stack-based algorithm which significantly improves performance over exhaustively computing trees and removing those whose root node is an ancestor of another from the answer set.

F. Summary and Discussion

The majority of keyword query languages discussed in this article target keyword-only querying of XML data. Few proposals address querying RDF data and several of them translate keyword queries into traditional query languages. On the other hand, XML keyword query languages for the most part are implemented as systems that evaluate the query without mapping it onto another query language.

However, most keyword query languages for XML limit themselves to processing tree-shaped data, that is, XML without any kind of hyperlinks. Those query languages that do work on graph-shaped XML, for example XRank, do not incorporate hyperlinks during the grouping of the matches. There is work on extending interconnection semantics to deal with XML data containing IDREF links [63] which due to its purely theoretical nature has not been discussed here. So far, to the best of the authors' knowledge, no keyword query language for graph-shaped XML that makes full use of the document links exists.

As [178] point out, one contributing reason for this is the expected increase in complexity and thus processing time which is detrimental in an application area dealing with large amounts of data.

Correspondingly, the lack of RDF keyword query languages that evaluate queries directly can be attributed to the fact that RDF is graph-shaped and cannot be easily converted into tree-shaped data like XML can. In addition, querying RDF poses the additional problems of treating labeled edges and blank nodes. One approach is to summarize the RDF graph into a different structure [166], [191], but this means that the structure of the data is partially ignored and the granularity of the query result is reduced.

In XML querying on the other hand, connecting or grouping matches is of great concern and a focus of many of the presented approaches. Several heuristics for grouping have been proposed as refinements to the established LCA concept, for example SLCA [196]), MLCA [133], CVLCA [132] and interconnection semantics [64]. All these approaches can be interpreted as extensions of LCA which add constraints in order to remedy the false positive problem of LCA and achieve improvement in grouping matched nodes according to their semantic entities. The difference between the algorithms for the computation of SLCA, VLCA etc. is thus the filter that they apply to remove undesirable results from the set of LCA nodes and, given a query and data, they produce a set of results that is a subset of the results obtained by applying LCA computation.

Determining semantic entities in structured data is important to keyword querying since, unlike in traditional query languages, queries are never fully specified –and indeed often do not allow the user to fully specify the query–, and consequently, the inferred semantics are used to retrieve informative results. While the approaches above determine the LCA, that is, the root node of the common entity of all keyword matches, based on keyword match instances, an alternative approach that was only used in the works discussed in this article in XKeyword [15], [111], but also mentioned in connection with XRank [106] and employed in keyword querying databases [27], [70] is to manually group the data into concepts and thus pre-define the possible query answer components. The difference between the two approaches is that the latter uses an extra level of processing where parts of query answers are defined a priori and therefore independent of a specific query. While this has the arguable disadvantage of requiring manual annotation, it foregoes two fundamental problems of LCA-based methods for automatic grouping:

The first is the underlying assumption that no element not in the subtree governed by the concept root node is relevant to the query answer. As mentioned in Section VI, this means that relevant information about an entity is not returned when the keywords in a query are contained in a subtree of the tree representing the entity. Returning to the example data in Figure 1, this means for instance that the queries $K = \{Doe, Smith\}$ and $K = \{Semantic, 2005\}$ will yield trivial results but not more information about the respective articles, for example the title and year of coauthored articles in the first case and the names of the authors in the second.

There are two different approaches to overcome this problem, namely displaying the query result in conjunction with the data and to enable search and browse behavior, or to allow matching on label nodes and enable a more targeted specification of a return value. Then, for example, the first keyword query could be extended to $K = \{Doe, Smith, title, journal\}$, meaning that the concept node (i.e. the root node of the semantic entity) is of type "article" and not "authors" and the entity subtree contains the desired information. This is possible in Cohen et al.'s query language [64] and XSeek [134], [135] and will be

discussed further below.

The first problem might be indicative of the second problem, namely that the different heuristics for grouping aim at being universal, or, at least, versatile solutions (which is why grouping is required in the first place, recall Amazon and their Advanced Search and uniform data structure mentioned in Section VI), but on the other hand appear to be data-driven solutions which make assumptions about the relations between structure and semantics which may not be universal. For example, the difference between data-centric and document-centric XML suggests different requirements concerning grouping and return values with multiple occurrences of the keywords within an XML subtree being indicative of particular relevance when document-centric XML is queried, which is not necessarily the case for data-centric XML. Consequently, all grouping strategies presented are not universally applicable and under certain circumstances lead to both false positives and false negatives. This observation raises the question to what extent it is even possible to reliably deduce semantics purely from structural characteristics of XML data.

In summary, manual grouping at the schema level performs well but has the obvious disadvantage that it requires users or administrators to invest time and effort to define the groupings. Manual grouping also has the advantage that data containing hyperlinks does not pose a problem. On the other hand, LCA and its variations are computed automatically but all algorithms rely on the presence of certain characteristics in the data to perform well.

Based on these observations, one way to achieve good grouping performance could be to simply consider the manual grouping as another, potentially optional, step of semantic annotation and to employ means to encourage users to perform the grouping.

A more promising solution is the use of modes to determine which grouping mechanism is appropriate given a certain data set or combination of query and data set. Since different assumptions about the relation between syntax and semantics in the data underlie the various grouping algorithms, the best algorithm could then be selected automatically, potentially leading to an improved overall performance.

To evaluate the feasibility of this approach, several questions have to be addressed: On the one hand, it is not clear how many –and which– grouping algorithms should be used, whether there is a universally optimal combination of grouping mechanisms or whether the selection should be application- and domain-dependent. Another, more basic, question is according to what characteristics the grouping mechanisms should be selected – it may be advantageous to use only a small number of maximally complementary algorithms to make the mode selection easier, but a combinations of a bigger number of algorithms could improve results since it could prove more versatile.

Another aspect are the questions which features or characteristics of the data or query should be considered to trigger a change of mode, how the optimal mode should be selected and, finally, how to combine the two, that is, create a mechanism that, given certain observed characteristics in the data, selects the preferred mode.

Possible features could be for example the amount of content relative to the amount of structural information, term frequency distributions and structural characteristics derived either from the schema or the data itself.

Learning, either through implicit or explicit feedback, could prove useful in arriving at a system which automatically selects the appropriate mode. Implicit feedback could for example be based on an analysis of which results are favored and disfavored by the users based on which results are clicked and which are skipped on a page of results [167]. Explicit feedback could be employed in the form of learning from a manually annotated training set or Query-By-Example type queries [202] where the user indicates the intended form of the result. Querying semi-structured data using the Query-By-Example paradigm has been researched previously, resulting in the query languages visXcerpt [23], [22] which both operate on XML data.

Even if automatic mode selection proves feasible, the issue of treating cyclic data remains since none of the automatic grouping mechanisms can operate on data containing hyperlinks. It is thus desirable to find a generalized universal grouping mechanism which can be applied both to XML and RDF data.

While connecting keyword matches is the focus in many of the keyword languages discussed, the form of the query answers themselves is less addressed. Strategies are for example to return the subtree governed by the concept node, the paths from the keyword matches to the concept nodes, or just the label of the concept node. These different return structures vary in their balance of the tradeoff between conciseness and information value. One important characteristic of traditional query languages, namely the targeted and flexible retrieval of elements, is present only in two of the presented stand-alone keyword query languages, namely Cohen et al.'s approach [64] and XSeek [134], [135]. In principle, the selection of the content of a node is realized in both through returning the content of a node whose label is matched. However, neither query language makes it possible to bind specific values to variables, and therefore it is not possible to further use query results in construct terms, as is a desirable feature in various applications, for example when embedding queries in Wiki pages.

Keyword-enhanced query languages, on the other hand, allow for more targeted selection and construction to varying degrees. Schmidt et al. [178] only retrieve the label of the LCA node, Florescu et al.'s approach [88] makes the granularity of the return value dependent on the specificity of the query and Schema-Free XQuery allows for the binding of variables to specific nodes in an entity subtree.

Unsurprisingly, keyword-enhanced query languages in general have greater expressiveness than the translation-

based languages and those implemented as stand-alone systems. Almost all of the translation keyword query languages have a very simple syntax with queries consisting only of keywords or label-keyword pairs. SemSearch, and Abbaci et al.'s [2] and Saito et al.'s [170] approaches are exceptions to this. SemSearch only offers a disjunction operator in addition to the implicit conjunction common to most other keyword query languages, while [170] allow for the use of some XPath operators. Among the stand-alone keyword query languages presented here, that in Abbaci et al.'s approach [2] has the most comprehensive query syntax with operators for disjunction, inclusion, parent-child relationship and negation. This query language is not directly concerned with grouping, but proposes a method for applying query operators by performing operations on sets of matched nodes that could potentially be combined with grouping heuristics to yield a query language that is both expressive and performs grouping while still having a simpler syntax than keyword-enhanced query languages.

Another focus in the area of keyword querying is the ranking of the results which in general is based on the principle that a smaller distance between matched nodes and between matched nodes and concept nodes means more specific and thus better results. Ranking usually is realized using the Vector Space Model and a variant of the tf-idf measure. It is advantageous to rank the results before fully generating them since this makes it possible to retrieve only the top k results, meaning that the results can be shown faster to the user and that processing time can be saved when the user is not interested in all results.

Another relevant issue is how to convey the vocabulary for queries. Keyword query languages are flexible with respect to the structure of the data and are the ability to query over heterogeneous data is emphasized as one of the advantages of keyword query languages. Heterogeneity can refer either to differences in the structural organization of the data or to differences in the vocabulary.

Figure 22 shows the data from Figure 1 in a different structural organization[20], namely here the articles are grouped by their authors. Due to the automatic grouping, one keyword query can be used to query both documents. However, the query results may of course differ since the grouping uses structural characteristics to find query answers.

Figure 23 displays another reformulation of the data in Figure 1 where the structure is identical but node labels differ. Queries involving node labels over 1 and 23 can never successfully retrieve results from both documents since they use different vocabularies. For example, the label-keyword queries $K_1 = \{published : 2005, surname : Smith\}$ and $K_2 = \{year : 2005, last : Smith\}$ express the same informational need, but do so in different words. Consequently, K_1 does not have any matches in the data in Figure 1, and the same is true for K_2 and 23. The problem is thus that a term can have many different synonyms and the user may not know which words to use to express her query. The problem also exists when data that is homogeneous with respect to their vocabulary are queried, since the user initially does not know which terms are used in the data, but is of particular concern when heterogeneous data using different vocabularies is queried, since there, no standardized vocabulary that the user could learn exists.

In a seminal study of the *vocabulary problem*, [93] found that participants used a big number of different terms are used to refer to the same concepts. The probability of two people choosing the same word for a given object was found to be below 20%. At most 36% of the participants chose the "best", that is, most frequent term for an object. The proposed solution to remedy this problem is to establish lists of synonyms or aliases for each term. For example, a system could map the term "published" to "year", thus enabling the use of both terms in queries.

Query expansion is thus applied to improve the recall in information retrieval applications through finding synonyms, morphological variations and misspellings. A variety of techniques for automatic query expansion have been proposed [69], [181], [72], [114], some of which are employed in the keyword query languages presented: Q2Semantic uses Wikipedia to find terms similar to the keywords. The authors of Schema-Free XQuery list several possibilities for obtaining a domain-specific thesaurus to be used with the "expand" function, namely deriving the synonyms for each terms from the corpus of XML data, creating it either manually or through information retrieval techniques like bootstrapping. If no domain-specific thesaurus is available, they suggest the use of a universal thesaurus, for example WordNet. This is also how *semantic mapping* functions in SPARK. In addition, *morphological mapping* is employed, which functions on the form (and not the semantics) of the keywords and uses stemming and other methods and measures from Natural Language Processing. Each term mapping is augmented with a confidence score, meaning that the list of synonyms can also serve as a controlled way to semantically relax the query.

Finally, one issue addressed in little of the presented works is the combined querying of RDF and XML data. SemSearch and QuizRDF can query Web documents augmented with RDF annotations, but the former only evaluates the query on the RDF annotations, meaning that it is not possible to impose conditions on both the document itself and its annotations in a query.

QuizRDF on the other hand allows to restrict the Web documents matching a given query through their RDF annotations. However, it is not possible to query the structure of the Web documents or combine XML and RDF search in a single query. Further, QuizRDF is a search-and-browse system and returns Web documents, that is, there is no

[20]The repetitions of the article subtrees are left out for reasons of legibility

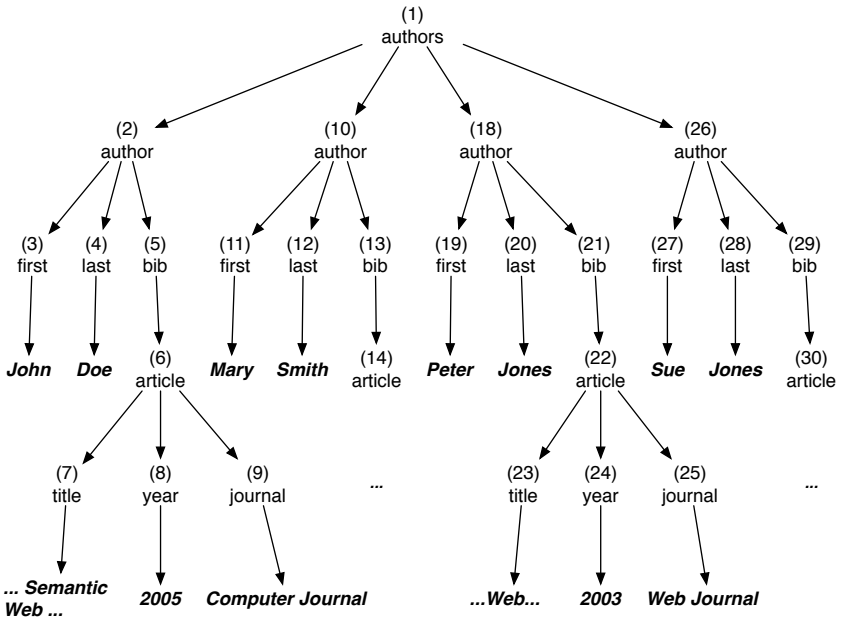

Fig. 22. Alternative XML-formalization of the data in 1

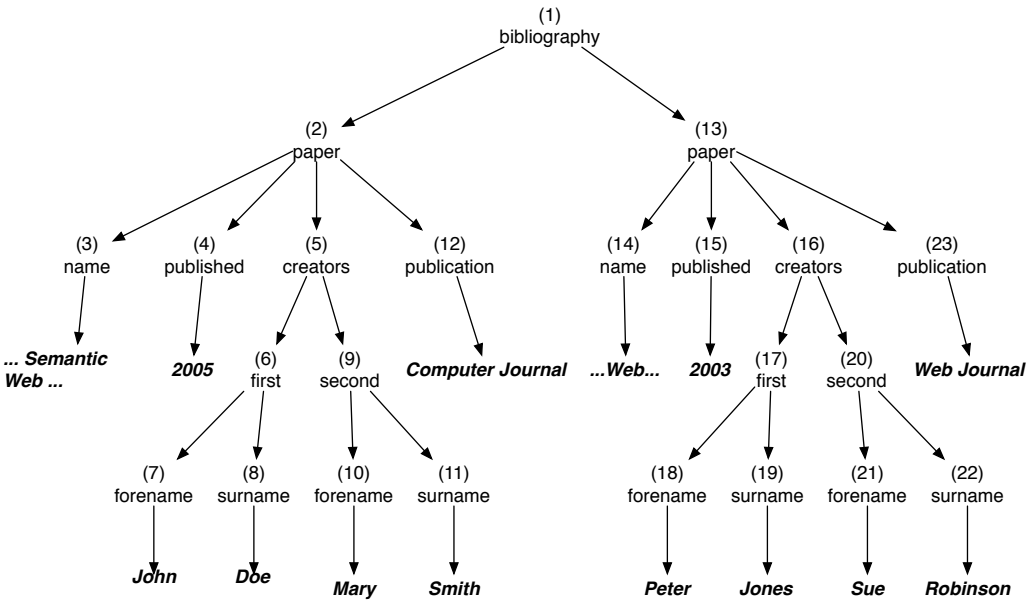

Fig. 23. Alternative XML-formalization of the data in 1

grouping of entities and consequently no flexible return values, meaning that the system is suited for interactive exploration of data rather than expressive querying at a high granularity.

The combined querying of XML and RDF is desirable in the context on the Semantic Web where not all content of the data (XML) is necessarily represented in metadata (e.g. RDF) and vice versa [28]. If querying of the two formalisms is possible using only one query language, recall is thus increased and further casual users only need to familiarize themselves with one query language, making the benefits of the Semantic Web accessible to a broad user base.

Integrated querying of RDF and XML is generally possible through serializing RDF into "flat triples" in XML, however this method is disadvantageous over the more natural view of RDF data as graphs [113], [91]. Integrated access to RDF and XML has been investigated, among others, in [169], [16], [160] and [91].

VII. Conclusion

With keyword-based query languages we have investigated one of the most promising and the, by far, best investigated approaches for combining the automation and deduction features of traditional, database-style Web queries with the ease-of-use of Web search.

As interesting and worthwhile as the comparison and classification of keyword-based query languages is in itself, in the context of the wider theme of this survey its most significant contribution is the relative lack of success in finding a universally applicable strategy for determining the extent of a meaningful answer to a keyword query. Considering the extent of research revealed by the comparison, this lack of success is all the more surprising and clear evidence that the central question of the dichotomy of Web search and Web queries remains unsolved: Can we find enable automation and deduction on the Web without loosing the accessibility for untrained operators?

In the presence of ever increasing data size, a positive answer for this questions becomes ever more essential for visions such as the Semantic Web. Though recent success of social or human-in-the-loop solutions for such problems (see Mechanical Turk[21] or the general field of human computation[22]) is certainly encouraging it is questionable how well such approaches scale and how far they can be generalized. In particular, where reliability and speed are issues automation, even if programmed painstakingly and at great cost, is unlikely to be replaced by social approaches. On the other hand, giving the vast numbers of untrained users in growing social networks on the Web just a little bit of automation may go further than sophisticated that is accessible only to few.

[21]http://www.mturk.com/mturk/welcome

[22]http://video.google.com/videoplay?docid=-8246463980976635143

Acknowledgements

The authors would like to thank James Bailey, Oliver Bolzer, Georg Gottlob, Ian Horrocks, Michael Kraus, Benedikt Linse, Renzo Orsini, Dimitris Plexousakis, and Sebastian Schaffert for many fruitful discussions on the topic of Web query languages, that culminated in two previous surveys [14], [92] on the topic.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 211932 (cf. http://www.kiwi-project.eu/) and from the European Commission and the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

References

[1] "iTQL Commands," Online only, 2004, http://www.kowari.org/271.htm.

[2] F. Abbaci, J. Valsamis, and P. Francq, "Index and Search XML Documents by Combining Content and Structure," *International Conference on Internet Computing, Las Vegas, Nevada, USA. June 26-29, 2006*, 2006.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wienerm, "The Lorel Query Language for Semistructured Data," *Intl. Journal on Digital Libraries*, vol. 1, no. 1, pp. 68–88, 1997.

[4] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases," in *Proc. ACM Symp. on Management of Data (SIGMOD)*. New York, NY, USA: ACM, 1989, pp. 253–262.

[5] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," in *Proc. Int. Conf. on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2002, p. 141.

[6] S. Amer-Yahia, R. Baeza-Yates, M. P. Consens, and M. Lalmas, "XML Retrieval: DB/IR in Theory, Web in Practice," in *Proceedings of the Interntional Conference on Very Large Databases (VLDB)*, 2007.

[7] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram, "XQuery and XPath Full Text 1.0," W3C, Candidate Recommendation, May 2008.

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

[9] S. Amer-Yahia and J. Shanmugasundaram, "XML Full-Text Search: Challenges and Opportunities," in *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2005.

[10] R. Angles and C. Gutierrez, "Querying RDF Data from a Graph Database Perspective," in *Proc. European Semantic Web Conf. (ESWC)*, ser. LNCS, vol. 3532, 2005.

[11] *plist — Property List Format*, Apple Inc., 2003.

[12] E. Augurusa, D. Braga, A. Campi, and S. Ceri, "Design and Implementation of a Graphical Interface to XQuery," in *Proc. Symposium of Applied Computing*. ACM Press, 2003, pp. 1163–1167.

[13] D. Backett, "Turtle—Terse RDF Triple Language," Institute for Learning and Research Technology, University of Bristol, Tech. Rep., 2007.

[14] J. Bailey, F. Bry, T. Furche, and S. Schaffert, "Web and Semantic Web Query Languages: A Survey," in *Reasoning Web: First International Summer School*, 2005.

[15] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, and T. Wang, "A System for Keyword Proximity Search on XML Databases," in *vldb'2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 2003, paper Yes, pp. 1069–1072.

[16] S. Battle, "Round-Tripping between XML and RDF," in *International Semantic Web Conference (ISWC)*, 2004.

[17] R. J. Bayardo, D. Gruhl, V. Josifovski, and J. Myllymaki, "An Evaluation of Binary XML Encoding Optimizations for fast Stream based XML Processing," in *Proc. Int. World Wide Web Conf.* ACM Press, 2004, pp. 345–354.

[18] D. Beckett and B. McBride, *RDF/XML Syntax Specification (Revised)*, Recommendation, W3C, 2004.

[19] M. Benedikt, W. Fan, and G. Kuper, "Structural Properties of XPath Fragments," in *Proc. International Conference on Database Theory*, 2003.

[20] M. Benedikt and C. Koch, "Interpreting Tree-to-Tree Queries," in *Proc. Int'l. Symp. on Automata, Languages and Programming (ICALP)*, 2006, pp. 552–564.

[21] ——, "XPath Leashed," in *ACM Computing Surveys*, 2007.

[22] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser, "Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web," *Proc. Int. Semantic Web Conf*, vol. 11, 2004.

[23] S. Berger, F. Bry, S. Schaffert, and C. Wieser, "Xcerpt and visXcerpt: from pattern-based to visual querying of XML and semistructured data," *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 1053–1056, 2003.

[24] S. Berger, F. Bry, O. Bolzer, T. Furche, S. Schaffert, and C. Wieser, "Querying the Standard and Semantic Web Using Xcerpt and Visxcerpt," in *Proceedings of European Semantic Web Conference (ESWC'05)*, Heraklion, Crete, Greece, 2005.

[25] S. Berger, F. Bry, T. Furche, and C. Wieser, "Visual Languages: A Matter of Style," in *Proceedings of the VLL 2007 workshop on Visual Languages and Logic*, Coeur d'Alene, Idaho, USA, 2007.

[26] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon, *XML Path Language (XPath) 2.0*, Working Draft, W3C, 2005.

[27] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 431–440, 2002.

[28] K. Bischoff, C. S. Firan, W. Nejdl, and R. Paiu, "Can All Tags Be Used for Search?" in *CIKM 2008: Proceedings of the 17th ACM Conference on Information and Knowledge Management*, Napa Valley, California, USA, 2008.

[29] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin, "Two-variable Logic on Data Trees and XML Reasoning," in *Proc. ACM Symp. on Principles of Database Systems (PODS)*. New York, NY, USA: ACM, 2006, pp. 10–19.

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

[31] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine," in *Proc. ACM Symp. on Management of Data (SIGMOD)*. New York, NY, USA: ACM Press, 2006, pp. 479–490.

[32] A. Bonifati, D. Braga, A. Campi, and S. Ceri, "Active XQuery," in *Proc. Int. Conf. on Data Engineering*. IEEE Computer Society, 2002, p. 403.

[33] T. Bray, D. Hollander, A. Layman, and R. Tobin, "Namespaces in XML (2nd Edition)," W3C, Recommendation, 2006.

[34] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Third Edition)," W3C, Recommendation, 2004.

[35] D. Brickley and R. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," W3C, Recommendation, 2004.

[36] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

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

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

[39] E. Bruno, J. L. Maitre, and E. Murisasco, "Extending XQuery with Transformation Operators," in *Proc. ACM symposium on Document Engineering*. ACM Press, 2003, pp. 1–8.

[40] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*. New York, NY, USA: ACM Press, 2002, pp. 310–321.

[41] F. Bry, T. Furche, B. Linse, and A. Schroeder, "Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs," in *Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM)*. ACM Press, 2006.

[42] 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*, vol. 1, no. 2, 2005.

[43] F. Bry, T. Furche, A. Hang, and B. Linse, "GRDDLing with Xcerpt: Learn one, get one free!" in *Proc. European Semantic Web Conf. (ESWC)*, 2007.

[44] F. Bry, T. Furche, C. Ley, B. Linse, and B. Marnette, "RDFLog: It's like Datalog for RDF," in *Proc. Workshop on (Constraint) Logic Programming (WLP)*, 2008.

[45] ——, "Taming Existence in RDF Querying," in *Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR)*, 2008.

[46] F. Bry, T. Furche, B. Linse, and A. Pohl, "XcerptRDF: A Pattern-based Answer to the Versatile Web Challenge," in *Proc. Workshop on (Constraint) Logic Programming (WLP)*, 2008.

[47] F. Bry and S. Schaffert, "A Gentle Introduction into Xcerpt, a Rule-Based Query and Transformation Language for XML," in *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on theSemantic Web*, 2002.

[48] R. G. G. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, Eds., *Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[49] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie, *XML Query Use Cases*, Working Draft, W3C, 2005.

[50] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie, *XML Query (XQuery) Requirements*, Working Draft, W3C, 2003.

[51] D. Chamberlin and J. Robie, "XQuery Update Facility Requirements," W3C, Working Draft, 2005.

[52] ——, "XQuery 1.1," W3C, Working Draft, 2008.

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

[54] L. Chen and E. A. Rundensteiner, "ACE-XQ: A CachE-aware XQuery Answering System," in *Proc. Workshop on the Web and Databases*, 2002.

[55] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based Algorithms for Pattern Matching on DAGs," in *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*. VLDB Endowment, 2005, pp. 493–504.

[56] T. Chen, J. Lu, and T. W. Ling, "On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*. New York, NY, USA: ACM Press, 2005, pp. 455–466.

[57] Z. Chen, H. V. Jagadish, L. V. Lakshmanan, and S. Paparizos, "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery," in *Proc. Int. Conf. on Very Large Databases*, 2003.

[58] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava, "Index Structures for Matching XML Twigs using Relational Query Processors," *Data & Knowledge Engineering (DKE)*, vol. 60, no. 2, pp. 283–302, 2007.

[59] V. Christophides, S. Cluet, and G. Moerkotte, "Evaluating Queries with Generalized Path Expressions," in *Proc. ACM SIGMOD International Conference on Management of Data*, 1996, pp. 413–422.

[60] 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.

[61] J. Clark, *XSL Transformations (XSLT) Version 1.0*, Recommendation, W3C, 1999.

[62] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and Distance Queries via 2-hop Labels," in *Proc. ACM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 937–946.

[63] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv, "Interconnection Semantics for Keyword Search in XML," *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pp. 389–396, 2005.

[64] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSearch: A Semantic Search Engine for XML," in *The 29th International Conference on Very Large Databases (VLDB)*, 2003, paper Yes.

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

[66] D. Connolly, "Gleaning Resource Descriptions from Dialects of Languages (GRDDL)," W3C, Recommendation, 2007.

[67] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," in *Proc. Int. Conf. on Very Large Databases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 341–350.

[68] J. Cowan and R. Tobin, "XML Information Set (2nd Ed.)," W3C, Recommendation, 2 2004.

[69] H. Cui, J. Wen, J. Nie, and W. Ma, "Probabilistic query expansion using query logs," *Proceedings of the 11th international conference on World Wide Web*, pp. 325–332, 2002.

[70] S. Dar, G. Entin, S. Geva, and E. Palmon, "DTL's DataSpot: Database exploration using plain language," *Proceedings of the 24rd International Conference on Very Large Data Bases*, pp. 645–649, 1998.

[71] J. Davies and R. Weeks, "QuizRDF: Search Technology for the Semantic Web," in *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, paper Yes, p. 40112.

[72] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[73] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu, "A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding," in *Proc. ACM SIGMOD Conf.* ACM Press, 2003, pp. 623–634.

[74] 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.

[75] ——, "A Query Language for XML," in *Proc. Int. World Wide Web Conf.*, 1999.

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

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

[78] P. F. Dietz, "Maintaining Order in a Linked List," in *Proc. ACM Symp. on Theory of Computing (STOC)*. New York, NY, USA: ACM, 1982, pp. 122–127.

[79] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics," W3C, Recommendation, 2 2007.

[80] A. Eisenberg and J. Melton, "An early Look at XQuery," *SIGMOD Record*, vol. 31, no. 4, pp. 113–120, 2002.

[81] ——, "An early Look at XQuery API for Java™(XQJ)," *SIGMOD Record*, vol. 33, no. 2, pp. 105–111, 2004.

[82] D. C. Fallside and P. Walmsley, "XML Schema Part 0: Primer Second Edition," W3C, Recommendation, 2004.

[83] P. Fankhauser, "XQuery Formal Semantics: State and Challenges," *SIGMOD Record*, vol. 30, no. 3, pp. 14–19, 2001.

[84] P. Fankhauser and P. Lehti, "XQuery by the book: The IPSI XQuery Demonstrator," in *XML Conference & Exhibition*, 2002.

[85] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model," W3C, Recommendation, 2007.

[86] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur, "Implementing XQuery 1.0 : The Galax Experience," in *Proc. Int. Conf. on Very Large Databases*, 2003.

[87] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan, "The BEA Streaming XQuery Processor," *VLDB Journal*, vol. 13, no. 3, pp. 294–315, 2004.

[88] D. Florescu, D. Kossmann, and I. Manolescu, "Integrating Keyword Search into XML Query Processing," in *Proceedings of the 9th International World Wide Web Conference on Computer networks : The International Journal of Computer and Telecommunications Networking*. Amsterdam, The Netherlands: North-Holland Publishing Co., 2000, paper Yes, pp. 119–135.

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

[90] T. Furche, "Implementation of Web Query Language Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost," Ph.D. dissertation, Ludwig-Maxmilians University Munich, 2008.

[91] T. Furche, F. Bry, and O. Bolzer, "XML Perspectives on RDF Querying: Towards Integrated Access to Data and Metadata on the Web," in *Grundlagen von Datenbanken 2005*, 2005.

[92] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob, "RDF Querying: Language Constructs and Evaluation Methods Compared," in *Reasoning Web, Second International Summer School 2006*, 2006.

[93] G. Furnas, T. Landauer, L. Gomez, and S. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[94] P. Genevès and J.-Y. Vion-Dury, "XPath Formal Semantics and Beyond: A Coq-Based Approach," in *Proc. Int'l. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*. Salt Lake City, Utah, United States: University Of Utah, August 2004, pp. 181–198.

[95] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," in *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 436–445.

[96] G. Gottlob and C. Koch, "Monadic Queries over Tree-Structured Data," in *Proc. Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, pp. 189–202.

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

[98] ——, "XPath Query Evaluation: Improving Time and Space Efficiency," in *Proc. International Conference on Data Engineering*, 2003.

[99] ——, "Efficient Algorithms for Processing XPath Queries," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 444–491, 2005.

[100] G. Gottlob, N. Leone, and F. Scarcello, "The Complexity of Acyclic Conjunctive Queries," *Journal of the ACM*, vol. 48, no. 3, pp. 431–498, 2001.

[101] T. Grust, "Accelerating XPath Location Steps," in *Proc. ACM Symp. on Management of Data (SIGMOD)*, 2002.

[102] T. Grust and J. Rittinger, "Jump Through Hoops to Grok the Loops — Pathfinder's Purely Relational Account of XQuery-style Iteration Semantics," in *Proc. ACM SIGMOD/PODS Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2008.

[103] T. Grust, J. Rittinger, and J. Teubner, "eXrQuy: Order Indifference in XQuery," in *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2007.

[104] T. Grust, M. V. Keulen, and J. Teubner, "Accelerating XPath Evaluation in any RDBMS," *ACM Transactions on Database Systems*, vol. 29, no. 1, pp. 91–131, 2004.

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

[106] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2003, paper Yes, pp. 16–27.

[107] C. Gutierrez, C. Hurtado, and A. O. Mendelzon, "Foundations of Semantic Web Databases," in *Proc. ACM Symp. on Principles of Database Systems (PODS)*. New York, NY, USA: ACM Press, 2004, pp. 95–106.

[108] D. Harel and R. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[109] P. Hayes and B. McBride, "RDF Semantics," W3C, Recommendation, 2004.

[110] J. Hidders, "Satisfiability of XPath Expressions," in *Int. Workshop on Databse Programming Languages*, 2003.

[111] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword Proximity Search on XML Graphs," in *Proceedings of the 19th International Conference on Data Engineering*, 2003, paper Yes.

[112] A. Hulgeri and C. Nakhe, "Keyword Searching and Browsing in Databases Using BANKS," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2002, paper Yes, p. 431.

[113] E. Hung, Y. Deng, and V. Subrahmanian, "RDF aggregate queries and views," *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 717–728, 2005.

[114] Y. Jing and W. Croft, "An association thesaurus for information retrieval," *Proceedings of RIAO*, vol. 94, no. 1994, pp. 146–160, 1994.

[115] K. Jones *et al.*, "A Statistical Interpretation of Term Specificity and Its Application in Retrieval," *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[116] 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.

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

[118] 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*, vol. 42, no. 5, pp. 617–640, August 2003.

[119] ——, "RQL: A Functional Query Language for RDF," in *The Functional Approach to Data Management*, P. Gray, P. King, and A. Poulovassilis, Eds. Springer-Verlag, 2004, ch. 18, pp. 435–465.

[120] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadler., *XQuery from the Experts: A Guide to the W3C XML Query Language*, 1st ed. Addison-Wesley, 8 2003.

[121] M. Kay, *XPath 2.0 Programmer's Reference*. John Wiley, 8 2004.

[122] ——, "XSL Transformations, Version 2.0," W3C, Recommendation, 2007.

[123] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong, "XSLT 2.0 and XQuery 1.0 Serialization," W3C, Working Draft, 2 2005.

[124] S. Kepser, "A Simple Proof of the Turing-Completeness of XSLT and XQuery," in *Proc. Extreme Markup Languages*, 2004.

[125] G. Klyne, J. J. Carroll, and B. McBride, "Resource Description Framework (RDF): Concepts and Abstract Syntax," W3C, Recommendation, 2004.

[126] C. Koch, "On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values," *tods*, vol. 31, no. 4, 2006.

[127] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier, "FluXQuery: An Optimizing XQuery Processor for Streaming XML Data," in *Proc. Int. Conf. on Very Large Databases*, 2004.

[128] L. Kong, R. Gilleron, and A. Lemay, "Retrieving Top Relaxed Tightest Fragments for XML Keyword Search," online, 2008.

[129] J. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[130] Y. Lei, V. Uren, and E. Motta, "SemSearch: A Search Engine for the Semantic Web," *Proceedings of the 5th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks, Lecture Notes in Computer Science*, pp. 238–245, 2006.

[131] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[132] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective Keyword Search for Valuable LCAs over XML Documents," in *CIKM '07: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*. New York, NY, USA: ACM, 2007, paper Yes, pp. 31–40.

[133] Y. Li, C. Yu, and H. V. Jagadish, "Schema-Free XQuery," in *VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*. VLDB Endowment, 2004, paper Yes, pp. 72–83.

[134] Z. Liu, J. Walker, and Y. Chen, "XSeek: A Semantic XML Search Engine Using Keywords," *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1330–1333, 2007.

[135] Z. Liu and Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2007, paper Yes, pp. 329–340.

[136] J. Lovins, "Development of a Stemming Algorithm." *Mechanical Translation and Computational Linguistics*, vol. 11, pp. 22–31, 1968.

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

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

[139] F. Manola, E. Miller, and B. McBride, "RDF Primer," W3C, Recommendation, 2004.

[140] J. Marsh, "XML Base," W3C, Recommendation, 2001.

[141] J. M. Martínez, "MPEG-7 Overview," INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO), Tech. Rep. ISO/IEC JTC1/SC29/WG11N6828, 2004.

[142] M. Marx, "Conditional XPath, the First Order Complete XPath Dialect," in *Proc. ACM Symposium on Principles of Database Systems*. ACM, 6 2004, pp. 13–22.

[143] ——, "XPath with Conditional Axis Relations," in *Proc. Extending Database Technology*, 2004.

[144] ——, "First Order Paths in Ordered Trees," in *Proc. Int'l. Conf. on Database Theory (ICDT)*, 2005, pp. 114–128.

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

[146] H. Meuss and K. U. Schulz, "Complete Answer Aggregates for Treelike Databases: a novel Approach to combine querying and navigation," *ACM Transactions on Information Systems*, vol. 19, no. 2, pp. 161–215, 2001.

[147] H. Meuss, K. U. Schulz, and F. Bry, "Towards Aggregated Answers for Semistructured Data," in *Proc. Int. Conf. on Database Theory*. Springer-Verlag, 2001, pp. 346–360.

[148] G. Miklau and D. Suciu, "Containment and Equivalence for an XPath Fragment," in *Proc. ACM Symposium on Principles of Database Systems*. ACM Press, 2002, pp. 65–76.

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

[150] S. Muñoz, J. Pérez, and C. Gutierrez, "Minimal Deductive Systems for RDF," in *Proc. European Semantic Web Conf. (ESWC)*, ser. Lecture Notes in Computer Science, vol. 4519. Springer Verlag, 2007, pp. 53–67.

[151] M. Murata, A. Tozawa, M. Kudo, and S. Hada, "XML Access Control using Static Analysis," in *Proc. ACM Conf. on Computer and Communications Security*. ACM Press, 2003, pp. 73–84.

[152] D. Olteanu, "SPEX: Streamed and Progressive Evaluation of XPath," *IEEE Transactions on Knowledge and Data Engineering*, 2007.

[153] D. Olteanu, T. Furche, and F. Bry, "An Efficient Single-Pass Query Evaluator for XML Data Streams," in *Data Streams Track, Proc. Symposium of Applied Computing*. ACM, 3 2004, i4.

[154] ——, "Evaluating Complex Queries against XML streams with Polynomial Combined Complexity," in *Proc. British National Conference on Databases*, 2004.

[155] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "XPath: Looking Forward," in *Proc. EDBT Workshop on XML-Based Data Management*, ser. Lecture Notes in Computer Science, vol. 2490. Springer Verlag, 2002.

[156] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORDPATHs: Insert-friendly XML Node Labels," in *Proc. ACM Symp. on Management of Data (SIGMOD)*. ACM Press, 2004, pp. 903–908.

[157] N. Onose and J. Simeon, "XQuery at your Web Service," in *Proc. Int. World Wide Web Conf.* ACM Press, 2004, pp. 603–611.

[158] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford University, Tech. Rep. SIDL-WP-1999-0120, 1999.

[159] R. Paige and R. E. Tarjan, "Three Partition Refinement Algorithms," *SIAM Journal of Computing*, vol. 16, no. 6, pp. 973–989, 1987.

[160] P. Patel-Schneider and J. Siméon, "The Yin/Yang Web: XML Syntax and RDF Semantics," in *WWW '02: Proceedings of the 11th international conference on World Wide Web*. New York, NY, USA: ACM, 2002, pp. 443–453.

[161] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," in *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2006.

[162] A. Polleres, "From SPARQL to Rules (and Back)," in *Proc. Int'l. World Wide Web Conf. (WWW)*. New York, NY, USA: ACM, 2007, pp. 787–796.

[163] A. Polleres, T. Krennwallner, J. Kopecky, and W. Akhtar, "XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage," in *Proc. European Semantic Web Conf. (ESWC)*, 2008.

[164] S. Pradhan, "An Algebraic Query Model for Effective and Efficient Retrieval of XML Fragments," *Proceedings of the 32nd International Conference on Very Large Data Bases*, pp. 295–306, 2006.

[165] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C, Proposed Recommendation, 2007.

[166] Y. Qu, "Q2RDF: Ranked Keyword Query on RDF Data," Southeast University, P.R. China, Tech. Rep., 2008.

[167] F. Radlinski and T. Joachims, "Query Chains: Learning to Rank from Implicit Feedback," in *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. New York, NY, USA: ACM, 2005, pp. 239–248.

[168] J. Robie, "Updates in XQuery," in *XML Conference & Exhibiton*, 2001.

[169] J. Robie, L. M. Garshol, S. Newcomb, M. Biezunski, M. Fuchs, L. Miller, D. Brickley, V. Christophides, and G. Karvounarakis, "The Syntactic Web," *Markup Lang.*, vol. 3, no. 4, pp. 411–440, 2001.

[170] T. Saito and S. Morishita, "Amoeba Join: Overcoming Structural Fluctuations in XML Data," *Proc. of WebDB, Chicago, USA*, pp. 38–43, 2006.

[171] G. Salton, A. Wong, and C. S. Yang, "A Vector Space Model for Automatic Indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

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

[173] S. Schaffert and F. Bry, "Querying the Web Reconsidered: A Practical Introduction to Xcerpt," in *Proc. Extreme Markup Languages (Int'l. Conf. on Markup Theory & Practice)*, 2004.

[174] S. Schaffert, R. Westenthaler, and A. Gruber, "Ikewiki: A User-Friendly Semantic Wiki," in *3rd European Semantic Web Conference (ESWC06)*, Budva, Montenegro, 2006.

[175] S. Schenk and S. Staab, "Networked Graphs: a Declarative Mechanism for SPARQL Rules, SPARQL Views and RDF Data Integration on the Web," in *Proc. Int'l. World Wide Web Conf. (WWW)*. New York, NY, USA: ACM, 2008, pp. 585–594.

[176] R. Schenkel, A. Theobald, and G. Weikum, "HOPI: An Efficient Connection Index for Complex XML Document Collections," in *Proc. Extending Database Technology*, 2004.

[177] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas, "Efficient Relational Storage and Retrieval of XML Documents," *International Workshop on the Web and Databases*, 2000.

[178] A. Schmidt, M. L. Kersten, and M. Windhouwer, "Querying XML Documents Made Easy: Nearest Concept Queries," in *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, paper Yes, pp. 321–329.

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

[180] D. W. Shipman, "The Functional Data Model and the Data Languages DAPLEX," *ACM Transactions on Database Systems*, vol. 6, no. 1, pp. 140–173, 1981.

[181] K. Sparck Jones, *Automatic keyword classification for information retrieval*. London, 1971.

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

[183] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 204–215, 2002.

[184] I. Tatarinov and A. Halevy, "Efficient Query Reformulation in peer Data Management Systems," in *Proc. ACM SIGMOD Conf.* ACM Press, 2004, pp. 539–550.

[185] S. Trißl and U. Leser, "Fast and Practical Indexing and Querying of Very Large Graphs," in *Proc. ACM Symp. on Management of Data (SIGMOD)*. New York, NY, USA: ACM, 2007, pp. 845–856.

[186] Z. Vagena, L. S. Colby, F. Özcan, A. Balmin, and Q. Li, "On the Effectiveness of Flexible Querying Heuristics for XML Data," in *XSym*, ser. Lecture Notes in Computer Science, 2007, pp. 77–91.

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

[188] N. Walsh and L. Muellner, *DocBook: The Definitive Guide*. O?Reilly, 10 1999.

[189] J. W. W. Wan and G. Dobbie, "Mining Association Rules from XML data using XQuery," in *Proc. Workshop on Australasian Information Security, Data Mining Web Intelligence, and Software Internationalisation*. Australian Computer Society, Inc., 2004, pp. 169–174.

[190] H. Wang, H. He2, J. Yang, P. S. Yu, and J. X. Yu, "Dual Labeling: Answering Graph Reachability Queries in Constant Time," in *Proc. Int'l. Conf. on Data Engineering (ICDE)*. Washington, DC, USA: IEEE Computer Society, 2006, p. 75.

[191] H. Wang, K. Zhang, Q. Liu, D. T. Tran, and Y. Yu, "Q2Semantic: A Lightweight Keyword Interface to Semantic Search," in *Proceedings of the 5th International Semantic Web Conference (ESWC'08)*, 2008, paper Yes.

[192] F. Weigel, "A Survey of Indexing Techniques for Semistructured Documents," Master's thesis, Institute for Informatics, University of Munich, http://www.pms.ifi.lmu.de/index.html#PA_Felix.Weigel, 2002.

[193] F. Weigel, K. U. Schulz, and H. Meuss, "The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations," in *Proc. Int'l. XML Database Symposium (XSym)*, ser. LNCS, vol. 3671. Springer-Verlag, 2005, pp. 49–67.

[194] N. Wiegand, "Investigating XQuery for Querying across Database Object Types," *SIGMOD Record*, vol. 31, no. 2, pp. 28–33, 2002.

[195] P. T. Wood, "On the Equivalence of XML Patterns," in *Proc. Int. Conf. on Computational Logic*. Springer-Verlag, 2000, pp. 1152–1166.

[196] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2005, paper Yes, pp. 527–538.

[197] C. Zaniolo, "The Database Language GEM," in *Proc. ACM SIGMOD Conf.*, 1983.

[198] X. Zhang, G. Cheng, and Y. Qu, "Ontology summarization based on RDF sentence graph," *Proceedings of the 16th international conference on World Wide Web*, pp. 707–716, 2007.

[199] X. Zhang, K. Dimitrova, L. Wang, M. E. Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner, "Rainbow: multi-XQuery Optimization using Materialized XML Views," in *Proc. ACM SIGMOD Conf.* ACM Press, 2003, pp. 671–671.

[200] X. Zhang, B. Pielech, and E. A. Rundesnteiner, "Honey, I shrunk the XQuery!: an XML Algebra Optimization Approach," in *Proc. International Workshop on Web Information and Data Management*. ACM Press, 2002, pp. 15–22.

[201] Q. Zhou, C. Wang, M. Xiong, H. Wang, and Y. Yu, "SPARK: Adapting Keyword Query to Semantic Search," *Proceedings of the 6th International Semantic Web Conference*, pp. 694–707, 2007.

[202] M. M. Zloof, "Query-by-Example: A Data Base Language," *IBM Systems Journal*, vol. 16, no. 4, pp. 324–343, 1977.