I4-D3

Development of Use Cases, Part I

Illustrating the Functionality of a Versatile Web Query Language

Abstract

For determining requirements and constructs appropriate for a Web query language, or in fact any language, use cases are of essence. The W3C has published two sets of use cases for XML and RDF query languages. In this article, solutions for these use cases are presented using Xcerpt. a novel Web and Semantic Web query language that combines access to standard Web data such as XML documents with access to Semantic Web meta-data such as RDF resource descriptions with reasoning abilities and rules familiar from logic-programming. To the best knowledge of the authors, this is the first in depth study of how to solve use cases for accessing XML and RDF in a single language: Integrated access to data and meta-data has been recognized by industry and academia as one of the key challenges in data processing for the next decade. This article is a contribution towards addressing this challenge by demonstrating along practical and recognized use cases the usefulness of reasoning abilities, rules, and semi-structured query languages for accessing both data (XML) and meta-data (RDF).

Keyword List
REWERSE,XML,RDF,Use Cases,Query,Query Language,W3C,Xcerpt

Development of Use Cases, Part I

Illustrating the Functionality of a Versatile Web Query Language

Oliver Bolzer · **François Bry** · **Tim Furche** · **Sebastian Kraus** · **Sebastian Schaffert**

Institute for Informatics, University of Munich
`http://pms.ifi.lmu.de/`

06 March 2004

Abstract

For determining requirements and constructs appropriate for a Web query language, or in fact any language, use cases are of essence. The W3C has published two sets of use cases for XML and RDF query languages. In this article, solutions for these use cases are presented using Xcerpt. a novel Web and Semantic Web query language that combines access to standard Web data such as XML documents with access to Semantic Web meta-data such as RDF resource descriptions with reasoning abilities and rules familiar from logic-programming. To the best knowledge of the authors, this is the first in depth study of how to solve use cases for accessing XML and RDF in a single language: Integrated access to data and meta-data has been recognized by industry and academia as one of the key challenges in data processing for the next decade. This article is a contribution towards addressing this challenge by demonstrating along practical and recognized use cases the usefulness of reasoning abilities, rules, and semi-structured query languages for accessing both data (XML) and meta-data (RDF).

Keyword List
REWERSE,XML,RDF,Use Cases,Query,Query Language,W3C,Xcerpt

Contents

Introduction

After a decade of experience with research proposals as well as standardized query languages for the conventional Web and following the recent emergence of query languages for the Semantic Web a reconsideration of design principles for Web and Semantic Web query languages is called for. In Bry *et al.* (2005) and Bry *et al.* (2004*a*) the authors of this article argued, that a new kind of query languages, called *versatile* query languages, is required for meeting the novel requirements arising in the advancing Web.

However, where Bry *et al.* (2005) laid out the design principles for such a query language, concrete use cases and functionalities for such query languages have not yet been investigated appropriately. Usage scenarios are provided in Badea *et al.* (2005), however aiming at an illustration of *where* versatile query languages are believed to be useful or even essential for efficient and effective data access. This article, on the other hand, focuses on a detailed description and illustration of concrete *functionality* asked for in a versatile query language.

To this end, data in two representation formalisms, viz. XML and RDF, that are common in the (standard and Semantic) Web are considered together with well-established collections of use cases for querying such data published by the W3C. Both collections of use cases are investigated from the perspective of a versatile query language. Where possible and appropriate, implementations of queries involved in the use cases are given in Xcerpt Schaffert and Bry (2004), a Web query language designed by the authors of this article. This choice reflects Xcerpt's unique position among Web query languages, as it is the first Web query language designed with versatility in mind and therefore enabling access to Web data represented in different formats such as XML, RDF, or Topic Maps.

The investigation of these two use case collections from the perspective of a versatile query language illuminates a number of essential observations about query languages for Web data:

· There is a considerable non-trivial overlap between the functionalities asked for in the two use case collections. Exemplary functionalities are, e.g., extraction queries (where a substructure of the input is extracted and returned) and the handling of irregular data (e.g., testing the existence or non-existence of some optional data).

· Interestingly, Clark (2004) describes a number of use cases, where access to RDF data is intertwined with XML processing, e.g., where the result of a query against RDF data is further transformed to yield XML output. Where the document suggests separate languages for the two tasks, we believe that a versatile query language where both RDF and XML data can be queried and transformed with the same constructs and concepts is easier to learn and gives rise to query programs that are easier to develop and maintain.

· In the use case collection proposed by Chamberlin *et al.* (2005), an entire section is devoted to use cases on data origination from a relational database using an XML query

1

language. Analogously, we believe that XML query languages should also consider data represented using RDF. This is only partially covered by the two use case collections considered here, as RDF triples can be represented as relational data. However, in Section 3.1 it is shown how XML query languages can be employed to provide convenient access to RDF data (under certain assumptions).

As further discussed in Sections 2.1.1 and 3.1.2, neither collection claims to cover all use cases for XML (resp. RDF) query languages. Certain aspects such as optional construction, views, injective queries, qualified descendant traversal (or conditional axes, cf. Marx (2004)) are not covered by the use case collection, some of them are discussed in the companion paper Badea *et al.* (2005). In this article, we concentrate on the use case collections as they are published and only shortly mention a list of further use cases for useful or, in some cases, even essential functionality not covered by the current collection.

Following, a short introduction into the query language Xcerpt in Chapter 1, the remainder of this work is divided in two parts: The first part, Chapter 2, discusses the collection of use cases for XML query languages proposed in Chamberlin *et al.* (2005). Following the structure of Chamberlin *et al.* (2005), the use cases are grouped into nine sections, however for providing a more natural flow of argument the order of the sections differs from Chamberlin *et al.* (2005). The ninth section from Chamberlin *et al.* (2005) is omitted, as Xcerpt's type system is still under development. The second part, Chapter 3 investigates the RDF data access use cases from Clark (2004). In contrast to Clark (2004), the use cases are further divided into five logical sections focusing on particular functionalities represented in the use cases. A short summary of the results of this work and further work on use cases not covered by the two collections that form the basis for Chapter 2 and 3 concludes the article (cf. Chapter 4).

Chapter 1

Xcerpt: A Versatile Web Query Language

An Xcerpt (for more detailed introductions see Schaffert and Bry (2004) and Schaffert (2004)\cite{xcerpt-eml,schaffert-thesis}) program consists of at least one *goal* and some (possibly zero) *rules*. Rules and goals contain query and construction patterns, called *terms*. Terms represent tree-like (or graph-like) structures. The children of a node may either be ordered, i.e., the order of occurrence is relevant (e.g., in an XML document representing a book), or unordered, i.e., the order of occurrence is irrelevant and may be chosen by the storage system (as is common in relational database systems). In the term syntax, an *ordered term specification* is denoted by square brackets **[ ]**, an *unordered term specification* by curly braces **{ }**.

Likewise, terms may use *partial term specifications* for representing incomplete query patterns and *total term specifications* for representing complete query patterns (or data items). A term t using a partial term specification for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. Partial term specification is denoted by *double* square brackets **[[ ]]** or curly braces **{{ }}**. In contrast, a term t using a total term specification does not match with terms that contain additional subterms without corresponding subterms in t. Total term specification is expressed using *single* square or curly braces. Matching is formally defined, e.g., in Schaffert (2004) using so-called *term simulation*.

Furthermore, terms may contain the *reference constructs* `^id` (*referring occurrence of the identifier* `id`) and `id @ t` (*defining occurrence of the identifier* `id`). Using reference constructs, terms can form cyclic (but rooted and directed) graph structures.

1.1 Data Terms

Data terms represent XML documents and the data items of a semistructured database, and may thus only contain total term specifications (i.e., single square brackets or curly braces). They are similar to *ground* functional programming expressions and logical atoms. A *database* is a (multi-)set of data terms (e.g., the Web). A non-XML syntax has been chosen for Xcerpt to improve readability, but there is a one-to-one correspondence between an XML document and a data term. Listings 1.1 and 1.2 give an impression of the Xcerpt term syntax. They

represent a train timetable (from `http://railways.com`) and a hotel reservation offer (from `http://hotels.net`).

Listing 1.1: Train timetable (at site `http://railways.com`)

```
 1  travel {
      last-changes-on { "2004-04-30" },
 3    currency { "EUR" },
      train {
 5      departure {
          station { "Munich" },
 7        date { "2004-05-03" },
          time { "15:25" }
 9      },
        arrival {
11        station { "Vienna" },
          date { "2004-05-03" },
13        time { "19:50" }
        },
15      price { "75" }
      },
17    train {
        departure {
19        station { "Munich" },
          date { "2004-05-03" },
21        time { "13:20" }
        },
23      arrival {
          station { "Salzburg" },
25        date { "2004-05-03" },
          time { "14:50" }
27      },
        price { "25" }
29    },
      train {
31      departure {
          station { "Salzburg" },
33        date { "2004-05-03" },
          time { "15:20" }
35      },
        arrival {
37        station { "Vienna" },
          date { "2004-05-03" },
39        time { "18:10" }
        }
41    }
      ...
43  }
```

Listing 1.2: Train timetable (at site `http://railways.com`)

```
    voyage {
 2    currency { "EUR" },
      hotels {
 4      city { "Vienna" },
        country { "Austria" },
 6      hotel {
          name { "Comfort Blautal" },
 8        category { "3 stars" },
          price-per-room { "55" },
10        phone { "+43 1 88 8219 213" },
          no-pets {}
12      },
        hotel {
14        name { "InterCity" },
          category { "3 stars" },
16        price-per-room { "57" },
          phone { "+43 1 82 8156 135" }
18      },
```

```
     hotel {
20      name { "Opera" },
        category { "4 stars" },
22      price-per-room { "106" },
        phone { "+43 1 77 8123 414" }
24    },
    ...
26  },
  ...
28 }
```

1.2 Query Terms

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

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

The following query term retrieves departure and arrival stations for a train in the train document. Partial term specifications (partial curly braces) are used since the train document might contain additional information irrelevant to the query.

Listing 1.3: Query term selecting departure and arrival stations>

```
1 travel {{
   train {{
3    departure {{
       station { var From } }},
5    arrival {{
       station { var To }   }}
7  }}
 }}
```

1.3 Construct Terms

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

The following construct term creates a summarized representation of trains grouped inside a trains term. Note the use of the **all** construct to collect all instances of the train subterm that

can be created from substitutions in the substitution set resulting from the query in Listing 1.3.

Listing 1.4: Construct term collecting train instances

```
1 trains {
    all train {
3     from { var From },
      to   { var To }
5   }
  }
```

1.4   Construct-Query Rules

Construct-query rules (short: rules) relate a construct term to a query consisting of **and** and/or **or** connected query terms. They have the following form:

```
1 CONSTRUCT
    ... construct term ...
3 FROM
    ... query term ...
5 END
```

Rules can be seen as *views* specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g., an XML document or a database). Queries or parts of a query may be further restricted by arithmetic constraints in a so-called condition box, beginning with the keyword **where**.

The following Xcerpt rule is used to gather information about the hotels in Vienna where a single room costs less than 70 Euro per night and where pets are allowed (specified using the **without** construct).

Listing 1.5: Rule for selecting and sorting sought-for hotels in Vienna

```
CONSTRUCT
2   answer [ all var H ordered by [ P ] ascending ]
  FROM
4   in {
      resource { "http://hotels.net" },
6     voyage {{
        hotels {{
8         city { "Vienna" },
          desc var H → hotel {{
10          price-per-room { var P },
            without no-pets {}
12        }}
        }}
14    }}
    } where { var P < 70 }
16 END
```

An Xcerpt query may contain one or several references to *resources*. Xcerpt rules may furthermore be *chained* like active or deductive database rules to form complex query programs, i.e., rules may query the results of other rules. Recursive chaining of rules is possible (but note that the declarative semantics in Schaffert (2004) requires certain restrictions on recursion). In contrast to the inherent structural recursion used, e.g., in XSLT, which is essentially limited to the tree structure of the input document, recursion in Xcerpt is always explicit and free in the sense that any kind of recursion can be implemented. Applications of recursion on the Web are manifold:

6

· Structural recursion over the input tree (like in XSLT) is necessary to perform transformations that preserve the overall document structure and change only certain things in arbitrary documents (e.g., replacing all em elements in HTML documents by strong elements).

· Recursion over the conceptual structure of the input data (e.g., over a sequence of elements) is used to iteratively compute data (e.g., create a hierarchical representation from flat structures with references).

· Recursion over references to external resources (hyperlinks) is desirable in applications like Web crawlers that recursively visit Web pages.

The following scenario illustrates the usage of a "conceptual" recursion to find train connections, including train changes, from Munich to Vienna.

The train relation (more precisely the XML element representing this relation) is defined as a view on the XML document seen as a database on trains):

Listing 1.6: Train relation: base case

```
1 CONSTRUCT
    train [ from [ var From ], to [ var To ] ]
3 FROM
    in {
5     resource { "file:travel.xml" },
      travel {{
7       train {{
          departure {{ station { var From } }},
9         arrival   {{ station { var To }   }}
        }}
11    }}
    }
13 END
```

A recursive rule implements the transitive closure train-connection of the relation train. If the connection is not direct (recursive case), then all intermediate stations are collected in the subterm via of the result. Otherwise, via is empty (base case).

Listing 1.7: Transitive closure of Listing 1.6

```
  CONSTRUCT
2   train-connection [
      from [ var From ],
4     to   [ var To ],
      via  [ var Via, all optional var OtherVia ]
6   ]
  FROM
8   and {
      train [ from [ var From ], to [ var Via ] ],
10    train-connection [
        from [ var Via ],
12      to   [ var To  ],
        via  [[ optional var OtherVia ]]
14    ]
    }
16 END

18 CONSTRUCT
    train-connection [
20    from [ var From ],
      to   [ var To ],
22    via  [ ]
    ]
```

7

```
24 FROM
     train [ from [ var From ], to [ var To ] ]
26 END
```

Based on the "generic" transitive closure defined above, the following rule retrieves only connections between Munich and Vienna.

Listing 1.8: Collecting all connections from Munich to Vienna

```
1 GOAL
     connections {
3      all var Conn
     }
5 FROM
     var Conn → train-connection [[ from { "Munich" } , to { "Vienna" } ]]
7 END
```

1.5 Accessing RDF Data

Xcerpt is not limited to XML data, but rather allows access to data in different formats. In Chapter 3 details on how to access RDF with Xcerpt are discussed along with the RDF use cases.

Chapter 2

Querying XML: Realizing the W3C XML Query Use Cases in Xcerpt

2.1 Introduction and Preliminaries

In this first section, the focus is on the XML Query use cases published by the W3C in Chamberlin *et al.* (2005). Implementations of most queries are given in Xcerpt and compared with the XQuery solutions given in Chamberlin *et al.* (2005).

Familiarity with XQuery is assumed for the rest of this article, cf. Boag *et al.* (2005) for the normative reference, Katz *et al.* (2003) for a practical introduction. Also in some parts, knowledge of the companion standards for XQuery, viz. XQuery's data model Fernández *et al.* (2005), XQuery's formal semantics Draper *et al.* (2005), XQuery's and XPath 2.0's functions and operator library Malhotra *et al.* (2005), are required.

2.1.1 The XML Query Use Cases Specification

To provide a forum for refinement and standardization of activities in industry and academia on querying of XML data, the W3C chartered the "XML Query" working group in 1999. In accordance to the working group charter, a first working draft providing use cases for XML query languages has been published in 2001. Where the initial draft Chamberlin *et al.* (2001) only contained queries (described in natural language) and sample data for the queries to operate on (given as DTD and/or exemplary XML document), later drafts also included solutions for the use cases in the XML query language developed by the working group, XQuery. Further updates clarified and completed the use cases. For this article, the version current at the time of writing Chamberlin *et al.* (2005) is used lasted changed in November 2003.

This current version of the "XML Query Use Cases" working draft Chamberlin *et al.* (2005) aims to "illustrate important applications for an XML query language [...] focused on specific application area[s]". To this end 77 queries spread over 7 use cases are described.

- For each of the 7 *use cases*, a natural language description, some sample data, and a schema for the data in form of a DTD or an XML schema is given. The latter one is required for some of the queries involving type information.

· For each of the 77 *queries*, a natural language specification of the query, the expected result of the query, and a solution in XQuery is given.

Although the collection of use cases and queries is broad and covers many of the expected features of an XML query language, it is, as stated in Chamberlin *et al.* (2005), "a snapshot of an ongoing work", where "some important application areas are not yet adequately covered by a use case". Some such application areas that the authers of this article deem essential for advanced Web query languages are presented in Badea *et al.* (2005). In our perception, the collection of use cases is also slightly biased with regard to the queries selected. It focuses mostly on functionality provided as part of XQuery, whereas equally important queries that are hard to express in XQuery, such as queries involving optionality, views, and injective queries (where, e.g., two distinct children of a node are to be queried), are not covered.

One particularly striking omission are use cases and queries on *graph-shaped data*, i.e., where additionally to the parent/child relation expressed by the nesting of XML elements non-hierarchical relations between elements in the document are used. Although XML provides for such non-hierarchical relations (by using ID/IDREF-links or other linking mechanisms such as XLink), current W3C data models (cf. DOM Apparao *et al.* (1998), XML Infoset Cowan and Tobin (2004), XQuery 1.0 and XPath 2.0 Data Model Fernández *et al.* (2005)) for XML consider XML tree-shaped and require such links to be traversed by an explicit join or similar means.

Nevertheless, the collection is sufficiently interesting and well-established to present a useful basis for assessing the functionality of a query language for querying XML data.

2.2 Core Functionality: Use Case "XMP--Experiences and Exemplars"

The focus of this use case and its accompanying queries is to demonstrate a number of essential queries "gathered from the database and document communities" Chamberlin *et al.* (2005). A number of queries are given that illustrate functionality that is expected to be useful for querying (Web) shops. The concrete scenario used is a book shop. Aside of the basic list of books (Listing 2.1), data about book reviews, possibly from another source (Listing 2.2), about prices of the books sold at different book store (Listing 2.3), and about the table of contents of some book (Listing 2.4) is available and used in the queries.

Listing 2.1: Basic List of Books (accessible as `http://bstore1.example.com/bib.xml`)

```
   <bib>
2      <book year="1994">
          <title>TCP/IP Illustrated</title>
4         <author><last>Stevens</last><first>W.</first></author>
          <publisher>Addison-Wesley</publisher>
6         <price> 65.95</price>
      </book>
8
      <book year="1992">
10        <title>Advanced Programming in the Unix environment</title>
          <author><last>Stevens</last><first>W.</first></author>
12        <publisher>Addison-Wesley</publisher>
          <price>65.95</price>
14     </book>

16     <book year="2000">
          <title>Data on the Web</title>
18        <author><last>Abiteboul</last><first>Serge</first></author>
```

```
          <author><last>Buneman</last><first>Peter</first></author>
20        <author><last>Suciu</last><first>Dan</first></author>
          <publisher>Morgan Kaufmann Publishers</publisher>
22        <price>39.95</price>
      </book>
24
      <book year="1999">
26        <title>The Economics of Technology and Content for Digital TV</title>
          <editor>
28                <last>Gerbarg</last><first>Darcy</first>
                  <affiliation>CITI</affiliation>
30        </editor>
              <publisher>Kluwer Academic Publishers</publisher>
32        <price>129.95</price>
      </book>
34
   </bib>
```

Notice, that this data set is rather flat and regular with the noticeable exception of information about editors and authors: A book can have an arbitrary number of authors and/or editors. In particular, the data is limited to depth four, no recursion in the data is allowed.

The second data set (cf. Listing 2.2) on reviews for books shares many of the characteristics of the first one: Again it is rather flat and, in this case, entirely regular (it could be straightforwardly be represented as a relational table without **null**-values).

Listing 2.2: Book Reviews (accessible as `reviews.xml`)

```
1  <reviews>
       <entry>
3          <title>Data on the Web</title>
           <price>34.95</price>
5          <review>
                   A very good discussion of semi-structured database
7                  systems and XML.
           </review>
9      </entry>
       <entry>
11         <title>Advanced Programming in the Unix environment</title>
           <price>65.95</price>
13         <review>
                   A clear and detailed discussion of UNIX programming.
15         </review>
       </entry>
17     <entry>
           <title>TCP/IP Illustrated</title>
19         <price>65.95</price>
           <review>
21                 One of the best books on TCP/IP.
           </review>
23     </entry>
   </reviews>
```

The same observations as for the previous data also applies for the third data set (cf. Listing 2.3). In this small example, only prices from two book shops, identified by their URLs (`bstore1.example.com` and `bstore1.example.com`) are presented.

Listing 2.3: Book Prices from multiple Book Stores (`http://bstore2.example.com/prices.xml`)

```
1  <prices>
       <book>
3          <title>Advanced Programming in the Unix environment</title>
           <source>bstore2.example.com</source>
5          <price>65.95</price>
       </book>
7      <book>
           <title>Advanced Programming in the Unix environment</title>
```

11

```
 9          <source>bstore1.example.com</source>
            <price>65.95</price>
11      </book>
        <book>
13          <title>TCP/IP Illustrated</title>
            <source>bstore2.example.com</source>
15          <price>65.95</price>
        </book>
17      <book>
            <title>TCP/IP Illustrated</title>
19          <source>bstore1.example.com</source>
            <price>65.95</price>
21      </book>
        <book>
23          <title>Data on the Web</title>
            <source>bstore2.example.com</source>
25          <price>34.95</price>
        </book>
27      <book>
            <title>Data on the Web</title>
29          <source>bstore1.example.com</source>
            <price>39.95</price>
31      </book>
    </prices>
```

Only the last data set (cf. Listing 2.4) is of a different nature: it is a typical exemplar for document-oriented XML (close to, e.g., the OASIS structured document standard DocBook Walsh and Muellner (1999)). Notice the recursive use of sections and of titles, which occur at different levels, in contrast to the elements in the previous data sets.

Listing 2.4: Table of Content for "Data Model" (accessible as `books.xml`)

```
 1 <chapter>
       <title>Data Model</title>
 3     <section>
           <title>Syntax For Data Model</title>
 5     </section>
       <section>
 7         <title>XML</title>
           <section>
 9             <title>Basic Syntax</title>
           </section>
11         <section>
               <title>XML and Semistructured Data</title>
13         </section>
       </section>
15 </chapter>
```

Based upon these data sets a number of queries are proposed and expected result as well as implementations in XQuery, where this promises additional insights, are given. In the following sections, solutions for these queries in Xcerpt are investigated and compared to the XQuery solutions from Chamberlin *et al.* (2005). The queries are grouped into logical sections that each focuses on elementary functionality of an XML query language. This makes this chapter an introduction to the functionality expected from an XML query language in general and to the realization of this functionality in Xcerpt.

2.2.1   XMP-Q1: Basic Selection

Query XMP-Q1:
List books published by Addison-Wesley after 1991, including their year and title.

The first query simply selects books based upon properties of the book (the properties realized as sub-elements of the book elements). It is worth noting, that the entire book element is expected as result, regardless of its actual content. This shows that the extraction of substructures of unknown size and shape (in this case, of the sub-tree rooted at a matching book element) is a natural and very basic feature of XML query languages. In contrast, queries in relational and most RDF query languages usually return flat (tabular) data of a fixed, known schema. This difference becomes more evident, when considering richly structured data as in Section 2.6.

As to be expected, such a basic query can be handily expressed in both Xcerpt and XQuery as shown in Listings 2.5 and 2.6.

Listing 2.5: XMP-Q1: Implementation in Xcerpt

```
   GOAL
 2 bib [
     all book [
 4     attributes  {
         year [ var Year ]
 6     },
       var Title
 8   ]
   ]
10 FROM
   in {
12   resource [ "http://bstore1.example.com/bib.xml", "xml" ],
     bib {{
14     book {{
         attributes  {{
16         year { var Year }
         }},
18       var Title → title {{}},
         publisher { "Addison−Wesley" }
20     }}
     }} where { var Year > 1991 }
22 }
   END
```

As discussed in Section [XXX Preliminaries], Xcerpt terms come in three variants: data terms (see [XXX]), query terms (see lines 11-21), and construct terms (see lines 2-9). The query term specifies a pattern for the data to be matched. Here, the data at the resource `http://bstore1.example.com/bib.xml` is matched against the patterns in lines 13-21: the data should be a single `bib` element containing a `book` element. The double braces in the term specification indicate that there might be multiple `book` elements and also further children of the `bib` element. Also, the use of curly braces indicates that we do not care about the order of the children of `bib` in this query. The `book` element is further required to have an attribute year, whose *value* is bound to the variable `Year`, a title child (that is bound to the variable `Title`), and a publisher child with value "Addison-Wesley". Finally, line 21 stipulates that the value of `Year` should be greater than 1991.

There are a few issues worth mentioning about this basic selection query:

- Since the query does not require a specific order among the elements inside the `bib` or `book` elements, unordered term specifications (indicated by curly brackets) are used. However, ordered term specifications (using square brackets) could in this case be used without changing the result, as it is known from the schema that the `title` child of a `book` always comes before the `publisher`. Given such schema information, the above query can be automatically rewritten to a query using ordered term specifications that might be more efficient to evaluate.

13

- Observe how Xcerpt allows a natural distinction between binding the value of an XML element or attribute (the `Year` variable) and binding an element including the sub-structure rooted at the element (the `Title` variable).

- Where the query term closely mimics the shape of the data to be matched, the construct term gives a good impression of the result of the Xcerpt program shown: Under a `bib` we will find for each selected book a `book` element containing year and title of the book (the year represented as an attribute).

- Finally, it is worth noting, that this query assumes that the query assumes that each book has exactly one title and year (as the schema given in Chamberlin *et al.* (2005)) specifies. Listing 2.9 shows an implementation for Query XMP-Q3 that can handle the case of several titles for one book: It is sufficient to simply add a grouping expression on the books found in the original data.

Contrast this solution for Query XMP-Q1 in Xcerpt to the following implementation using XQuery (found in Chamberlin *et al.* (2005)):

Listing 2.6: XMP-Q1: Implementation in XQuery

```
1  <bib>
   {
3    for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
     where $b/publisher = "Addison-Wesley" and $b/@year > 1991
5    return
       <book year="{ $b/@year }">
7        { $b/title }
       </book>
9  }
   </bib>
```

The XQuery solution is, as in the case of Xcerpt, fairly straightforward: a `bib` element is created that contains the result of an XQuery FLWOR expression. The FLWOR expression iterates over all `bib/book` elements (line 3). For each such element it is (a) tested whether one of its `publisher` children has the string value "Addison-Wesley" and whether its year attribute has a value greater than 1991 (line 4) and (b) if that test succeeds a `book` element with `year` attribute is created containing all `title` children of the queried `book` element.

Observe the existential quantification implicitly expressed by using a so-called general comparison (cf. Boag *et al.* (2005)). In this query both = and > express general comparisons. Given the schema of the data, it would have been possible to use value comparisons (`eq` and `gt`) instead.

If one compares the two implementations, a few differences and similarities are striking:

- Using patterns for specifying the data to be matched in the style of the "query-by-example" paradigm (Zloof 1975, Zloof (1977)) seems even more natural for hierarchical data than for relational databases for which it has originally been envisioned. Where queries expressed in XQuery often use patterns for constructions (sometimes referred to as "fill-in-the-blanks" programming style), Xcerpt uses patterns for both querying and construction.

- Maybe the most obvious difference is the clear separation of query and construction in the case of Xcerpt, which are intermingled in the XQuery solution (thereby producing a more compact solution). Combined with the use of patterns for querying *and* construction, this makes it very easy to understand the structure of the queried as well as the

constructed data in the Xcerpt case. For XQuery this is less obvious. Where the shape of the constructed data can be fairly easily grasped in this simple example, that is less so for the shape of the queried data. It is, for instance, not easy to recognize that the XQuery solution (in a very particular interpretation of query XMP-Q1) actually does not require that there is any title child for a book matched, however does require that there is a year attribute (by its use in the **where** clause). In the Xcerpt solution both are evidently required, as the natural language formulation of query XMP-Q1 seems to indicate.

· In both implementations, an explicit path from the document element to the selected data is specified although both languages provide constructs for traversing arbitrary paths (XQuery: **descendant** axis, Xcerpt: **desc** construct). This is typical for querying data with rather regular structure.

Most of the issues raised during the discussion of this first example can be similarly be observed in many of the following ones, however will not be further discussed in the remaining descriptions of queries and solutions.

2.2.2 XMP-Q2,XMP-Q3,XMP-Q4: Basic Grouping

A natural requirement for query languages is the ability to group data. The following two queries illustrate that grouping in hierarchical data is often based on structure rather than on values of the data queried.

Query XMP-Q2:
Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element.

This query is the base case for selecting two (or more) related information pieces. It can be seen as a form of projection, as from the structure only the relation between authors and titles (by virtue of the enclosing book element) is retained and represented by nesting each such pair in a separate result.

The following program presents an implementation of this query in Xcerpt: in the query term two variables Title and Author are used to select all title and author elements. In the construct term, the **all** construct is used to create one result element for each pair of binding of the two variables.

Listing 2.7: XMP-Q2: Implementation in Xcerpt

```
   GOAL
 2 results [
     all result [
 4     var Title,
       var Author
 6   ]
   ]
 8 FROM
   in {
10   resource [ "http://bstore1.example.com/bib.xml", "xml" ],
     bib {{
12     book {{
         var Title → title {{}},
14       var Author → author {{}}
       }}
16   }}
   }
18 END
```

15

Listing 2.8: XMP-Q2: Implementation in XQuery

```
  <results>
2   {
      for $b in doc("http://bstore1.example.com/bib.xml")/bib/book,
4         $t in $b/title,
          $a in $b/author
6     return
          <result>
8             { $t }
              { $a }
10        </result>
  }
12 </results>
```

Comparing this Xcerpt implementation to the XQuery implementation from Chamberlin *et al.* (2005)shown in Listing 2.8, illustrates that the **all** in Xcerpt is akin to a (nested) **for** loop in XQuery, where result is generated only in the inner loop and the loop variables correspond to the Xcerpt variables directly nested inside the **all** ("free" variables w.r.t. this **all**).

Observing this correspondence, one might then question why there is an additional **for** clause looping over all books in the XQuery solution. Indeed, this clause actually leads to a unintuitive behavior of the XQuery solution: Assume that two books have the same title *T* and a common author *A*. Then the XQuery solutions actually generates *two* result elements for the pair (T, A), a behavior that is not solicited by the formulation of Query XMP-Q2. Changing this behavior requires an explicit use of, e.g., the `distinct-values` function on the sequence of `result` elements or a similar change in the query for avoiding duplicate pairs.

Summarizing this shows two interesting differences between Xcerpt and XQuery:

· In many cases XQuery requires explicit duplicate removal, that is the default behavior of Xcerpt. Actually, the XPath (2.0) sub-language of XQuery also requires duplicate avoidance or removal: Compare, e.g., `for $t in //section//title/text() return $t` on the data from Listing 2.4 with `for $b in //section, $t in $b//title/text() return $t`. Where the former returns the text of titles that occur under several sections once only, the latter duplicates them (see also Section 4.2 in Draper *et al.* (2005), where it is shown that path expressions are normalized to a properly ordered, duplicate-free sequence).

· The interaction and differences between **for** (loop) expressions and path expressions in XQuery are arguably obscure and hard to grasp for a query programmer.

Returning to the discussion of basic grouping functionality in XML query languages, the next query basically extends XMP-Q2 by the requirement to group the selected titles and authors by the books they belong to.

Query XMP-Q3:
For each book in the bibliography, list the title and authors, grouped inside a "result" element.

The Xcerpt implementation of this query is actually surprisingly close to the implementation of XMP-Q2. The main difference is in the need to bind `book` elements to the variable `Book` and to use this variable for grouping the bindings for `Title` and `Author`. Notice also the additional nested **all**s that make it obvious that for each `result` element there can now be several titles and authors (in contrast to the implementation of XMP-Q2).

16

```
    GOAL
 2  results [
      all result [
 4      all var Title,
        all var Author
 6    ] group by { var Book }
    ]
 8  FROM
    in {
10    resource [ "http://bstore1.example.com/bib.xml", "xml" ],
      bib {{
12      var Book → book {{
          var Title → title {{}},
14        var Author → author {{}}
        }}
16    }}
    }
18  END
```

Again, we contrast this to the XQuery implementation shown in Listing 2.10. Here, we see that the nested loops are dropped and instead the selection of titles and authors is moved inside the construction of the result element. While this may lead to a very compact query, it is hard to grasp that this query does no longer construct pairs of exactly one title and exactly one author, but rather that there may be several title and author elements nested in a single result.

Listing 2.10: XMP-Q3: Implementation in XQuery

```
    <results>
 2  {
        for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
 4      return
            <result>
 6              { $b/title }
                { $b/author  }
 8          </result>
    }
10  </results>
```

The final query in this sections expands on the previous ones by grouping all titles for books of an author:

Query XMP-Q4:
For each author in the bibliography, list the author's name and the titles of all books by that author, grouped inside a "result" element.

Again, the Xcerpt implementation is very similar to the implementations of the previous queries. The nesting of the **all** constructs allows to gather all titles for each author.

Listing 2.11: XMP-Q4: Implementation in Xcerpt

```
    GOAL
 2  results [
      all result [
 4      var Author,
        all var Title
 6    ]
    ]
 8  FROM
    in {
10    resource [ "http://bstore1.example.com/bib.xml", "xml" ],
      bib {{
```

```
12    book {{
         var Title → title {{}},
14       var Author → author {{}}
      }}
16   }}
  }
18 END
```

The XQuery implementation on the other hand is considerably more involved than the implementation for the previous queries. The main reason for this is the need to use `distinct-values` for avoiding duplicate authors (as might be preferable in the implementation of XMP-Q1 too) and the split of `author` elements in `first` and `last`. The latter is required for avoiding duplicate authors in the result, as XQuery (or more precisely the XQuery function collection Malhotra *et al.* (2005)) only provides `distinct-values` for duplicate removal which is based on value equality (`eq` operator). For this example, however a different equality represented by the `deep-equal` function in XQuery (and used as standard equality in Xcerpt) is required. This (structural) equality is similar to term equality in logic programming languages.

Listing 2.12: XMP-Q4: Implementation in XQuery

```
  <results>
2   {
      let $a := doc("http://bstore1.example.com/bib/bib.xml")//author
4     for $last in distinct-values($a/last),
          $first in distinct-values($a[last=$last]/first)
6     order by $last, $first
      return
8         <result>
              <author>
10               <last>{ $last }</last>
                 <first>{ $first }</first>
12            </author>
              {
14               for $b in doc("http://bstore1.example.com/bib.xml")/bib/book
                 where some $ba in $b/author
16                   satisfies ($ba/last = $last and $ba/first=$first)
                 return $b/title
18            }
          </result>
20   }
  </results>
```

From the previous examples it should be evident, that Xcerpt provides a flexible and powerful, yet easy to use grouping mechanism. This is essential for an XML query language as grouping queries are central to (re-) constructing hierarchical data and therefore event more common than when considering relational data.

2.2.3 XMP-Q5: Basic Value-based Joins

Where the previous two sections show the importance of selection and grouping, this section focuses on *value-based* joins. We emphasize the value-based here, as it is arguable that already the previous queries used a form of joins when requiring that, e.g., two elements are children of the same parent. However, this section shows queries using explicit joins.

Query XMP-Q5:
For each book found at both bstore1.example.com and bstore2.example.com, list the title of the book and its price from each source.

18

Listing 2.13: XMP-Q5: Implementation in Xcerpt

```
1  CONSTRUCT
   books-with-prices [
3    all book-with-prices [
       var T,
5      price-bstore1 [ var Pa ],
       price-bstore2 [ var Pb ]
7    ]
   ]
9  FROM
   and {
11   in {
       resource { "http://bstore1.example.com/bib.xml", "xml" },
13     bib [[
         book [[
15         var T → title {{}},
           price [ var Pa ]
17       ]]
       ]]
19   },
     in {
21     resource { "http://bstore2.example.com/reviews.xml", "xml" },
       reviews [[
23       entry [[
           var T → title {{}},
25         price [ var Pb ]
         ]]
27     ]]
     }
29 }
   END
```

The implementation of this query is rather straightforward in both Xcerpt and XQuery as shown in Listings 2.13 and 2.14. Xcerpt uses a logical conjunction (expressed using **and**) of two query terms that are matched against the two data sets. The variable T occurs in both data terms as the matched `title` element, thereby restricting matches to those combinations where the `book` and the `entry` elements have a common `title` sub-element. I.e., multiple occurrences of the same variable inside the same query term or in query terms connected by **and** correspond to equi-joins.

In the XQuery case, a general comparison between the titles from bstore1 and the titles from bstore2 is used, i.e., it is tested whether at least one of the titles of a book from bstore1 occurs also as title in an entry from bstore2. Since the data set limits the number of titles to one, this comparison is sufficient in this case.

Listing 2.14: XMP-Q5: Implementation in XQuery

```
1  <books-with-prices>
     {
3      for $b in doc("http://bstore1.example.com/bib.xml")//book,
           $a in doc("http://bstore2.example.com/reviews.xml")//entry
5      where $b/title = $a/title
       return
7          <book-with-prices>
               { $b/title }
9              <price-bstore2>{ $a/price/text() }</price-bstore2>
               <price-bstore1>{ $b/price/text() }</price-bstore1>
11         </book-with-prices>
     }
13 </books-with-prices>
```

The previous example shows that value-based joins can be expressed easy enough in both XQuery and Xcerpt. Where Xcerpt uses a logic-programming style term equality (similar to XQuery's `deep-equal`), that is often very natural when joining structured data (see also Query

19

XMP-Q4), XQuery offers a number of different equality operators (based upon typed value of an element, based upon identity, based on term equality) some of which can be used to compare sequences.

2.2.4 XMP-Q6,XMP-Q7: Ordered Querying

Since XML represents ordered trees, querying the order of elements in the tree is an important requirement for an XML query language. The following two queries illustrate this requirement.

Query XMP-Q6:
For each book that has at least one author, list the title and first two authors, and an empty et-al element if the book has additional authors.

XMP-Q6 poses two challenges that have not occurred in previous queries: (a) only the *first* two authors should be returned (this requires the ability to query the order of the authors and a means to filter selected data based on order) and (b) the number of authors conditions the shape of the result (adding an empty et-al element).

The Xcerpt solution shown in Listing 2.15 is the first exemplar of an Xcerpt program using not only one **GOAL** rule, but rather several rules that are chained when evaluating the query:

· Rule 1 (lines 45–62) annotates each book element in the source data with the number of its authors (using the **count** aggregation function). Notice that here already books with no authors are excluded as asked for by the query (line 58). This annotation is, in contrast to the XQuery case shown below, necessary in Xcerpt, as aggregation requires sets of bindings and is therefore only available in construct terms.

· In typical logic-programming style, two rules are used for expressing an alternative: Rules 2 (lines 28–43) and 3 (lines 10–26) implement the two alternative shapes based upon the number of authors: If the number of authors is two or less, rule 2 matches and constructs a book containing the title and all (as there are at most two) authors of the book. If on the other hand there are more than 2 authors, rule 3 matches and only returns the first two authors but additionally an empty et-al element. Notice the use of the **first** keyword in line 14. In contrast to **all**, it limits the number of bindings for Author to be considered to the first two. If no further order is specified (using **order by**) the document order is used. The conditions specified in the **where** clause (or "condition box") are mutually exclusive and covering thereby guaranteeing that each book element from the source data matches with exactly one of the two rules.

· Finally, the goal (lines 1–8) collects the book elements generated by either Rule 2 or Rule 3. The order in which the books will occur inside bib is not specify and might vary depending on the implementation. If one would like to enforce a particular order, a **order by** clause would have to be added in line 4.

Listing 2.15: XMP-Q6: Implementation in Xcerpt

```
1  % Goal
   GOAL
3    bib {
       all var Book
5    }
   FROM
```

```
 7    var Book → book {{}}
   END
 9
   % Rule No. 3
11 CONSTRUCT
      book {
13      var Title,
        first 2 var Author,
15      et_al []
      }
17 FROM
      book_with_authorcount [
19      book [[
          var Title → title {{}},
21        var Author → author {{}}
        ]],
23      authorcount { var Count }
      ]
25    where { var Count > 2 }
   END
27
   % Rule No. 2
29 CONSTRUCT
      book {
31      var Title,
        all var Author
33    }
   FROM
35    book_with_authorcount [
      book [[
37      var Title → title {{}},
        var Author → author {{}}
39    ]],
      authorcount { var Count }
41    ]
      where { var Count <= 2 }
43 END

45 % Rule No. 1
   CONSTRUCT
47    all book_with_authorcount [
        var Book,
49      authorcount {
          count ( all var Author )
51      }
      ]
53 FROM
     in {
55      resource { "http://bstore1.example.com/bib.xml", "xml" },
        bib [[
57        var Book → book {{
            var Author → author {{}}
59        }}
        ]]
61    }
   END
```

In this case, the XQuery solution (cf. Listing 2.16) is noticeably more compact due to the intermingling of construction and querying: the number of authors can be used directly in the **where** clause. Furthermore, using nested queries and a selection construct **if-then-else** in lines 13–15 further shrinks the size of the query. Notice also, that lines 8–11 could be further reduced to { $b/author[position()<=2] }.

Listing 2.16: XMP-Q6: Implementation in XQuery

```
1 <bib>
   {
3    for $b in doc("http://bstore1.example.com/bib.xml")//book
```

```
     where count($b/author) > 0
5    return
         <book>
7            { $b/title }
             {
9                for $a in $b/author[position()<=2]
                 return $a
11           }
             {
13               if (count($b/author) > 2)
                  then <et-al/>
15                else ()
             }
17       </book>
     }
19 </bib>
```

Where the above query required the ability to query the order in which elements occur in the source data, the second query in this section demonstrates the use of order when constructing data, as known from, e.g., SQL. Both the query and the solutions prove to be straightforward extensions of Query XMP-Q1 and its solutions.

Query XMP-Q7:
List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order.

The following solution in Xcerpt is almost identical to Listing 2.5, the Xcerpt solution for Query XMP-Q1. The only change s the addition of the **order by** clause in line 8. It specified that the constructed book elements are to be ordered by the bindings of the Title variable using normal lexical order of characters (in contrast to XQuery internationalisation and support for different character orders has not yet been considered for Xcerpt). **lexical** could have been omitted, as this is the default order.

Listing 2.17: XMP-Q7: Implementation in Xcerpt

```
1  CONSTRUCT
   bib [
3    all book [
       attributes  {
5        year { var Year }
       },
7      title { var Title }
     ] order by [Title] lexical
9  ]
   FROM
11 in {
     resource { "http://bstore1.example.com/bib.xml", "xml" },
13   bib {{
       var Book → book {{
15       attributes  {
           year { var Year }
17       },
         title { var Title },
19       publisher { "Addison−Wesley" }
       }}
21   }} where { var Year > 1991 }
   }
23 END
```

The XQuery implementation is similarly a direct extension of its solution for Query XMP-Q1 by an **order by** clause.

```
<bib>
2  {
     for $b in doc("http://bstore1.example.com/bib.xml")//book
4    where $b/publisher = "Addison-Wesley" and $b/@year > 1991
     order by $b/title
6    return
         <book>
8            { $b/@year }
             { $b/title }
10       </book>
   }
12 </bib>
```

2.2.5 XMP-Q8,XMP-Q9: Querying Character Data

The following two queries showcase the need for matching character data (e.g., the text contained in elements, their labels, etc.) in XML query languages.

Query XMP-Q8:
Find books in which the name of some element ends with the string "or" and the same element contains the string "Suciu" somewhere in its content. For each such book, return the title and the qualifying element.

When considering semi-structured data it is a common observation that the knowledge about the structure of the data but also about its content is very limited. Therefore, queries like XMP-Q8, where only some information about the label or content of queried elements is known actually occur frequently in practice.

Listing 2.19: XMP-Q8: Implementation in Xcerpt

```
CONSTRUCT
2 bib [
    all book [
4     var Title,
      var X
6   ]
  ]
8 FROM
  in {
10  resource { "http://bstore1.example.com/bib.xml", "xml" },
    bib [[
12    book {{
        var Title → title {{}},
14      desc var X → /.*or/ {{
          desc /.*Suciu.*/
16      }}
      }}
18   ]]
  }
20 END
```

Xcerpt provides POSIX-style regular expressions for matching character data. Thanks to the convenient term syntax, regular expressions can be used directly where usually an element label or string occurs. Line 14 specified that the element label of a binding for X must match with the regular expression /.*or/, i.e., any number of arbitrary characters followed by "or". Line 15 further specifies that some text node under the binding for X contains "Suciu" again using a regular expression. Notice, the use of the **desc** keyword to express that neither the

23

level of the element X is bound to nor the text node matters as long as the text node occurs inside the element.

The XQuery solution shown in the following Listing actually assumes that only direct children of a book qualify (see line 2, where the element is selected using the (implicit) **child** axes). In contrast to Xcerpt, XQuery does not allow the use of regular expressions (or similar constructs) directly instead of a label node test. Therefore, the query selects all elements (wildcard label node test) and restricts the label only inside the predicate. XQuery provides two specialized functions for testing containment of one string in another (`contains`) and for testing whether one strings end in another one (`local-name`). Equivalently, one could use XQuery's regular expression facility (`fn:matches(string(.), "Suciu")` and `fn:matches(local-name(.), "or$")`). Note that Xcerpt, in contrast to XQuery, is implicitly anchored, i.e., a regular expression is matched against an entire string whereas in XQuery a regular expression is matched against all substrings of the string unless explicitly anchored by ^ (start of string/line) or $ (end of string/line). Therefore the XQuery regular expression `or$` matches the same strings as the Xcerpt regular expression `.*or`.

One striking difference is the `string` function used in the XQuery soliution, that returns the concatenated value of all text descendants of a node. In Xcerpt this has to be expressed specifically using a **desc** construct. This also points to a limitation of the Xcerpt solution: there the string "Suciu" must entriely be contained in a single element, whereas the XQuery solution could also cover a case such as `<initial>S</inital>uciu`. In Xcerpt, the values have to be explicitly concatenated to obtain the same effect.

Listing 2.20: XMP-Q8: Implementation in XQuery

```
1  for $b in doc("http://bstore1.example.com/bib.xml")//book
   let $e := $b/*[contains(string(.), "Suciu")
3               and ends-with(local-name(.), "or")]
   where exists($e)
5  return
       <book>
7          { $b/title }
           { $e }
9      </book>
```

As the previous query, the following one does not restrict the level at which the element searched for occurs.

Query XMP-Q9:
In the document `books.xml`, *find all section or chapter titles that contain the word "XML", regardless of the level of nesting.*

This query is a typical query against document-oriented XML and should be expressible easily in any XML query language. Indeed both the Xcerpt and XQuery solution are straightforward and rather similar. The main difference is that the Xcerpt solution uses a regular expression over the label whereas the XQuery solution uses a union of two sequences (expressed using |). The union operator has the advantage that it can be used also if the axes would differ (e.g., to select both chapter children and section descendants), where Xcerpt would require a disjunction as shown in Section 2.2.7.

Listing 2.21: XMP-Q9: Implementation in Xcerpt

```
1  CONSTRUCT
   results [
3    all var Title
```

```
     ]
 5   FROM
     in {
 7     resource { "books.xml", "xml" },
       desc /chapter|section/ [[
 9         var Title → title [[ /.*XML.*/ ]]
       ]],
11     }
     }
13   END
```

```
   <results>
 2   {
       for $t in doc("books.xml")//(chapter | section)/title
 4     where contains($t/text(), "XML")
       return $t
 6   }
   </results>
```

2.2.6 XMP-Q10: Basic Aggregation

Having considered aggregation over the structure of an XML document in Query XMP-Q6 (counting the number of author elements), the following query requires aggregation over element values.

Query XMP-Q10:
In the document `prices.xml`, find the minimum price for each book, in the form of a `minprice` element with the book title as its title attribute.

Listing 2.23: XMP-Q10: Implementation in Xcerpt

```
   CONSTRUCT
 2 results [
     all minprice [
 4     price {
         attributes  {
 6         title { var T }
         },
 8       min ( all var Price )
       }
10   ]
   ]
12 FROM
   in {
14   resource { "prices.xml", "xml" },
     prices [[
16     book [[
         title [ var T ],
18       price [ var Price ]
       ]]
20   ]]
   }
22 END
```

Both implementations are fairly straightforward. In contrast to Query XMP-Q6 shows the XQuery solution actually the effort to separate the querying (in lines 3–5) from the construction in the **return** clause. This shows that such, often desirable, separation can to some extent also be achieved in XQuery, whereas it is actually enforced in Xcerpt.

25

```
1  <results>
     {
3      let $doc := doc("prices.xml")
       for $t in distinct-values($doc//book/title)
5      let $p := $doc//book[title = $t]/price
       return
7        <minprice title="{ $t }">
           <price>{ min($p) }</price>
9        </minprice>
     }
11 </results>
```

2.2.7 XMP-Q11: Conditional Construction

Query XMP-Q11:
For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor's affiliation.

This query clearly illustrates an observation from Query XMP-Q6: Conditional construction, i.e., construction where depending on some condition on the input data different results are constructed, is realized in Xcerpt by multiple rules and chaining, whereas XQuery uses nested queries and/or **if-then-else** expressions.

Listing 2.25: XMP-Q11: Implementation in Xcerpt

```
   GOAL
2    bib [ all var Book  ]
   FROM
4    or {
       var Book → book {{}},
6      var Book → reference {{}}
     }
8  END

10 CONSTRUCT
     book [
12     var Title,
       all var Author
14   ]
   FROM
16   in {
       resource { "file:xmp−bib.xml", "xml" },
18     bib [[
         book [[
20         var Title → title {{}},
           var Author → author {{}}
22       ]]
       ]]
24   }
   END

26
   CONSTRUCT
28   reference [
       var Title,
30     affiliation [ var Affiliation ]
     ]
32 FROM
     in {
34     resource { "file:xmp−bib.xml", "xml" },
       bib [[
36       book [[
           var Title → title {{}},
```

26

```
38        editor [[
            affiliation [[ var Affiliation ]]
40        ]]
        ]]
42    ]]
    }
44 END
```

```
1  <bib>
   {
3        for $b in doc("http://bstore1.example.com/bib.xml")//book[author]
         return
5            <book>
                 { $b/title }
7                { $b/author }
             </book>
9  }
   {
11       for $b in doc("http://bstore1.example.com/bib.xml")//book[editor]
         return
13         <reference>
             { $b/title }
15           {$b/editor/affiliation}
           </reference>
17 }
   </bib>
```

2.2.8 XMP-Q12: Set Joins

Query XMP-Q12:
Find pairs of books that have different titles but the same set of authors (possibly in a different order).

As argued in query XMP-4, joining based on logic-programming style term equality, where not only the value of an element but also its structure and, recursively, the structure of its subelements is considered, is often required for XML data.

Listing 2.27: XMP-Q12: Implementation in Xcerpt

```
   GOAL
2  bib {
     all bookpair {
4      title { var Title1 },
       title { var Title2 }
6    }
   }
8  FROM
   temp [[
10   title_authorset {{
       title { var Title1 },
12     var Author_set → author_set {{}}
     }},
14   title_authorset {{
       title { var Title2 },
16     var Author_set → author_set {{}}
     }}
18 ]] where { var Title1 != Title2 }
   END
20
   CONSTRUCT
22 temp [
```

27

```
        all title_authorset {
24        title { var Title },
          author_set { all var Author }
26      }
    ]
28 FROM
    in {
30    resource { "http://bstore1.example.com/bib.xml", "xml" },
      bib [[
32      book {{
          title { var Title },
34        var Author → author {{}}
        }}
36    ]]
    }
38 END
```

The implementation in Xcerpt makes use of its ability to construct data terms where the order of the children does not matter (specified using curly brackets). When comparing two such terms, they are considered equal even if the actual order of their children differ. Using this ability of Xcerpt the implementation of the query is fairly simple: In the rule shown in lines 20–37 for each book a tuple consisting in its title and all its authors grouped inside an author_set element is created. Note the use of curly brackets to indicate that the order of the children of the author_set does not matter in comparisons and matching. In lines 1–18, this data is further queried such as to find pairs of such tuples with same author_set. Notice, that to avoid duplicate pairs an order among the tuples is enforced by using square brackets when querying the enclosing temp wrapper element.

Contrast this to the XQuery solution from Chamberlin *et al.* (2005) shown in the following Listing.

Listing 2.28: XMP-Q12: Implementation in XQuery

```
  <bib>
2 {
      for $book1 in doc("http://bstore1.example.com/bib.xml")//book,
4        $book2 in doc("http://bstore1.example.com/bib.xml")//book
      let $aut1 := for $a in $book1/author
6                  order by $a/last, $a/first
                   return $a
8     let $aut2 := for $a in $book2/author
                   order by $a/last, $a/first
10                 return $a
      where $book1 << $book2
12    and not($book1/title = $book2/title)
      and deep-equal($aut1, $aut2)
14    return
          <book-pair>
16            { $book1/title }
              { $book2/title }
18        </book-pair>
  }
20 </bib>
```

Here, for each pair of books two nested queries are used to obtain the ordered sequence of authors for each book and then the two sequences are compared using deep-equal (cf. Query XMP-Q4). As in the Xcerpt case, duplicates are avoided by requiring an order among the two books, cf. line 11. In contrast to Xcerpt it is necessary in line 12 to explicitly state inequality between the two books. In Xcerpt, this is guaranteed since two siblings are assumed to be different if specified in the same term.

2.3 Tree Extraction: Use Case "TREE--Queries that preserve Hierarchy"

Starting with this section, the following discussions will focus on queries against data with varying characteristics. In particular queries are considered that highlight interesting or typical requirements for XML and Web query languages in general.

The previous queries have mostly operated on rather flat and regular data. In this section a first example of richly structured data with high depth and recursive structure definition (where elements with same label can occur nested), cf. Listing 2.29, an excerpt from an XML representation of some book, sed for the queries in the remainder of this section.

Listing 2.29: Excerpt of a book (accessible at `book-content.xml`)

```xml
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section id="intro" difficulty="easy" >
    <title>Introduction</title>
    <p>Text ... </p>
    <section>
      <title>Audience</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Web Data and the Two Cultures</title>
      <p>Text ... </p>
      <figure height="400" width="400">
        <title>Traditional client/server architecture</title>
        <image source="csarch.gif"/>
      </figure>
      <p>Text ... </p>
    </section>
  </section>
  <section id="syntax" difficulty="medium" >
    <title>A Syntax For Data</title>
    <p>Text ... </p>
    <figure height="200" width="500">
      <title>Graph representations of structures</title>
      <image source="graphs.gif"/>
    </figure>
    <p>Text ... </p>
    <section>
      <title>Base Types</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Representing Relational Databases</title>
      <p>Text ... </p>
      <figure height="250" width="400">
        <title>Examples of Relations</title>
        <image source="relations.gif"/>
      </figure>
    </section>
    <section>
      <title>Representing Object Databases</title>
      <p>Text ... </p>
    </section>
  </section>
</book>
```

2.3.1 TREE-Q1,TREE-Q6: Tree Transformations

The two queries in this section are representatives of typical transformation queries, where the structure of the result reflects to some extent the structure of the input but changes such as renaming of elements, leaving out intermediary elements or entire subtrees, or adding envelopes around elements may occur. Typically, such queries require a recursive traversal of the structure of the input data, constructing the result data for each element based upon the result for its children. Whereas such recursive traversal has to be explicitly specified by the query author in both Xcerpt and XQuery, XSLT, the W3C's transformation language for XML data, is tailored to this kind of queries and therefore provides this recursive traversal as part of its evaluation model. To demonstrate the ease of writing such queries in XSLT, a solution for Query TREE-Q1 is also provided in XSLT.

> **Query TREE-Q1:**
> *Prepare a (nested) table of contents for "Data on the Web", listing all the sections and their titles. Preserve the original attributes of each section element, if any.*

Listings 2.30 and 2.31 show solutions for this query in Xcerpt and XQuery. Both solutions explicitly recur over the structure of the input data.

Consider first the Xcerpt solution: It uses three rules to map from the input book to the required output table of contents. The rules construct pairs of elements (inside book2toc elements) mapping elements from the input to result elements wrapping the corresponding output elements in the table of contents that is to be generated. Depending on the input element, there are two cases for this mapping:

· *Mapping section elements:* Rule 2 (lines 18–42) constructs an entry in the table of contents for a section element from the input. Notice, the use of the **optional** for handling in one rule elements with or without attributes and with or without additional child elements beyond the section title. For each such additional child, the corresponding entries in the table of contents are generated recursively (see line 40).

· *Mapping all other elements:* For all elements except sections, Rule 3 (lines 44–62) applies (as the expression !/section/ matches all element labels that do not match the regular expression /section/. As these elements do not require a map entry, the rule simply recurs over their children, if there are any (again realized using **optional**). This rule handles the case where section elements are nested inside elements that are not sections.

Listing 2.30: TREE-Q1: Implementation in Xcerpt

```
1  % Goal
   GOAL
3    toc [
         all var Toc
5    ]
   FROM
7    and {
       in {
9        resource { "book-content.xml", "xml" },
         book {{
11         var TopSection → section {{ }}
       }}
13     },
       book2toc[ var TopSection, result[ var Toc ] ]
15   }
   END
```

```
17
   % Rule No. 2
19 CONSTRUCT
     book2toc [
21     var Section,
       result [
23       section [
           optional var Attributes,
25         var Title,
           optional all var ChildToc
27       ]
       ]
29   ]
   FROM
31   and {
       in {
33       resource { "book−content.xml", "xml" },
         var Section → desc section {{
35         optional var Attributes → attributes {{ }},
           var Title → title {{ }},
37         optional var Child → /.*/ {{ }}
         }}
39     },
       optional book2toc [var Child, result [ var ChildToc ]]
41   }
   END

43
   % Rule No. 1
45 CONSTRUCT
     book2toc [
47     var Section,
       result [
49       optional all var ChildToc
       ]
51   ]
   FROM
53   and {
       in {
55       resource { "book−content.xml", "xml" },
         var Element → desc !/section/ {{
57         optional var Child → /.*/ {{ }}
         }}
59     },
       optional book2toc [var Child, result [ var ChildToc ]]
61   }
   END
```

In contrast to the above Xcerpt solution, the following XQuery program assumes that sections occur continuously under the root. In other words, it does not treat the case where a section is nested inside anything but another section or the book document element. However this case could also be handled, e.g., by adding an conditional expression to local:toc with only a recursion over its children in the case of non-section elements.

Apart of this, the XQuery solution uses a recursive function to obtain much the same effect as the Xcerpt one, with the difference that it does not require sections to have title children.

Since both cases do not occur in the actual data (and are actually not allowed by its schema as shown in Chamberlin *et al.* (2005)) the result of both solutions if applied to the data from Listing 2.29 is the same.

Listing 2.31: TREE-Q1: Implementation in XQuery

```
1 declare function local:toc($book-or-section as element()) as element()*
  {
3     for $section in $book-or-section/section
      return
5       <section>
```

```
         { $section/@* , $section/title , local:toc($section) }
7      </section>
   };

9
   <toc>
11    {
       for $s in doc("book−content.xml")/book return local:toc($s)
13    }
   </toc>
```

As stated before, XSLT is tailored to this kind of queries and indeed the XSLT solution shown in the following Listing is strikingly simple.

As pointed out above, the main reason for this is the implicit recursive traversal of the input structure when using **apply-templates**.

Listing 2.32: TREE-Q1: Implementation in XSLT

```
   <?xml version="1.0" encoding="utf−8"?>
2  <xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

4
     <xsl:template match="book">
6      <toc>
         <xsl:apply-templates />
8      </toc>
     </xsl:template>

10
     <xsl:template match="*">
12      <xsl:apply-templates />
     </xsl:template>

14
     <xsl:template match="section[title]">
16      <section>
         <xsl:copy-of select="@*" />
18        <xsl:copy-of select="title" />
         <xsl:apply-templates />
20      </section>
     </xsl:template>

22
   </xsl:stylesheet>
```

The first template (lines 5–9) matches the document element of the input (a book element) and transforms it into a toc element that contains the result of applying the templates defined in the XSLT program upon the children of the book element.

The second template (lines 11–13) specifies that all elements for which no more specific template is given should simply be skipped and the processing continued with their children.

Finally, the third template (lines 15–21) handles section elements (if they have a title child) by copying all attributes and title children and then processing its children.

Query TREE-Q6 shown in the following is very similar to this one. Actually it only differs in the requirement to also include the number of figures immediately contained in the section.

> **Query TREE-Q6:**
> *Make a nested list of the section elements in "Data on the Web", preserving their original attributes and hierarchy. Inside each section element, include the title of the section and an element that includes the number of figures immediately contained in the section.*

This additional information can be added easily to the above solutions. In the Xcerpt case one adds before line 37 an (optional) variable binding for figure children:

```
       optional var Figure → figure {{ }},
```

In the same rule before line 26, the count of these bindings is returned inside a figcount element:

```
    figcount [
2     count( all optional var Figure )
    ],
```

For XQuery the change is equally easy.

2.3.2 TREE-Q2,TREE-Q5: Flattening Tree Structures

Where the previous two queries showed how to *preserve* the structure, the following two ones require *flattening*. Both queries are very straightforward and (except for the need to query elements at arbitrary depth) similar to the ones in Section 2.2.

Query TREE-Q2:
Prepare a (flat) figure list for "Data on the Web", listing all the figures and their titles. Preserve the original attributes of each figure element, if any.

Listing 2.33: TREE-Q2: Implementation in Xcerpt

```
  GOAL
2 figlist [
    all figure {
4     attributes  { all optional var Attributes },
      var Title
6   }
  ]
8 FROM
  in {
10   resource { "file:book-tree.xml", "xml" },
    desc figure {{
12     attributes  { optional var Attributes },
      var Title → title {{}}
14   }}
  }
16 END
```

Listing 2.34: TREE-Q2: Implementation in XQuery

```
1 <figlist>
    {
3     for $f in doc("book.xml")//figure
      return
5        <figure>
            { $f/@* }
7            { $f/title }
        </figure>
9   }
  </figlist>
```

As stated above, both implementations are obvious and rather concise. Notice the use of **desc** in the Xcerpt and // (i.e., /descendant-or-self::node()/) in XQuery for selecting figure elements at arbitrary depth.

Solutions for Query TREE-Q5 are very similar to the previous ones, but additionally require as in the case of Query TREE-Q6 aggregation over the figure children. For brevity, the XQuery solution is omitted (but can of course found in Chamberlin *et al.* (2005)).

33

Query TREE-Q5:

Make a flat list of the `section` elements in "Data on the Web". In place of its original attributes, each `section` element should have two attributes, containing the title of the section and the number of figures immediately contained in the section.

Listing 2.35: TREE-Q5: Implementation in Xcerpt

```
1  GOAL
   section_list {
3    all section {
       attributes  {
5        title { var Title },
         figcount {
7          count ( all optional var Figure )
         }
9      }
     }
11 }
   FROM
13 in {
     resource { "file:book−tree.xml", "xml" },
15   desc section {{
       title { var Title },
17     optional var Figure → figure {{}}
     }}
19 }
   END
```

As in Query TREE-Q6, the `count` aggregation function is used to find the number of figures in a section. As in Query TREE-Q2, sections at arbitrary depth are matched.

2.3.3 TREE-Q3,TREE-Q4: Aggregation

This section is concluded by two more examples on structure aggregation. Both queries show variations of previously encountered aggregations.

Query TREE-Q3:

How many sections are in "Data on the Web", and how many figures?

Listing 2.36: TREE-Q3: Implementation in Xcerpt

```
1  CONSTRUCT
   result [
3    section_count {
       count ( all optional var Section )
5    },
     figure_count {
7      count ( all optional var Figure )
     }
9  ]
   FROM
11 in {
     resource { "file:book−tree.xml", "xml" },
13   and {
       desc optional var Section → section {{}},
15     desc optional var Figure → figure {{}}
     }
17 }
   END
```

The novelty in this query is the need to aggregate over elements that are not siblings, as the number of all sections and figures regardless of their position in the document is to be

reported. Xcerpt and XQuery can use the same means for aggregation in both cases. Here, XQuery's solution is very concise but also differs noticeably from the solution for, e.g., TREE-Q6 where the number of figures per section are counted.

Listing 2.37: TREE-Q3: Implementation in XQuery

```
1  <section_count>{ count(doc("book.xml")//section) }</section_count>,
   <figure_count>{ count(doc("book.xml")//figure) }</figure_count>
```

Query TREE-Q4:
How many top-level sections are in "Data on the Web"?

Again solutions for this query are straightforward and familiar from previous queries.

Listing 2.38: TREE-Q4: Implementation in Xcerpt

```
   CONSTRUCT
2  top_section_count [
     count ( all optional var Topsection )
4  ]
   FROM
6  in {
     resource { "file:book−tree.xml", "xml" },
8    book {{
       optional var Topsection → section {{}}
10   }}
   }
12 END
```

Listing 2.39: TREE-Q4: Implementation in XQuery

```
1  <top_section_count>
    {
3     count(doc("book.xml")/book/section)
    }
5  </top_section_count>
```

2.4 Ordered Data: Use Case "SEQ--Queries based on Sequence"

In this section, the ability of query languages to access data based upon its position is more closely investigated (cf. also Section 2.2.4). In particular, both the absolute position of data in the document (e.g., among its siblings or in document order) and the relative position of elements (e.g., what occurs before some element? what is in between two elements?) is considered in the following sections.

All queries in this section are based upon the following sample data, a medical report using the HL7 Patient Record Architecture (cf. Alschuler *et al.* (2000)). It is a description of some medical procedure where the description test is tagged with elements describing activities (e.g., anesthesia, incision). Notice that many elements, in particular section.content contain mixed content, i.e., character data and elements interspersed. Also most elements can occur at different levels in the document.

Listing 2.40: Data set illustrating a report on a medical procedure (identified by report1.xml)

```
1  <report>
     <section>
3      <section.title>Procedure</section.title>
```

```
      <section.content>
 5    The patient was taken to the operating room where she was placed
      in supine position and
 7    <anesthesia>induced under general anesthesia.</anesthesia>
      <prep>
 9      <action>A Foley catheter was placed to decompress the bladder</action>
      and the abdomen was then prepped and draped in sterile fashion.
11    </prep>
      <incision>
13      A curvilinear incision was made
        <geography>in the midline immediately infraumbilical</geography>
15      and the subcutaneous tissue was divided
        <instrument>using electrocautery.</instrument>
17    </incision>
      The fascia was identified and
19    <action>#2 0 Maxon stay sutures were placed on each side of the midline.
      </action>
21    <incision>
        The fascia was divided using
23      <instrument>electrocautery</instrument>
        and the peritoneum was entered.
25    </incision>
      <observation>The small bowel was identified.</observation>
27    and
      <action>
29      the
        <instrument>Hasson trocar</instrument>
31      was placed under direct visualization.
      </action>
33    <action>
        The
35      <instrument>trocar</instrument>
        was secured to the fascia using the stay sutures.
37    </action>
      </section.content>
39  </section>
    </report>
```

2.4.1 SEQ-Q1,SEQ-Q2: Querying Absolute Position

The first two queries focus on querying data based upon its position among its siblings or in the document.

Query SEQ-Q1:
In the Procedure section of `report1.xml`*, what Instruments were used in the second Incision?*

In the Xcerpt and XQuery solutions shown in this section you will notice a clear difference in how absolute positions are queried:

- In line with its focus on patterns, Xcerpt provides direct means (via **position**) to query the absolute position of an element among all of its children, i.e., its position insider a pattern. However, it is not possible to directly query the position of an element among any other set of elements except its siblings, e.g., among all children with the same label or all descendants. For this, it is necessary to gather all those elements in one rule as siblings in a newly constructed term. From this term, the sought for element can than be extracted.

- XQuery on the other hand is based upon sequences. This allows the direct appliance of positional constructs to any sequence of elements queried by an XQuery expression.

This difference in style is illustrated by the solutions for the first query.

Listing 2.41: SEQ-Q1: Implementation in Xcerpt

```
1  GOAL
   result {
3    all var Instrument
   }
5  FROM
   incisions [
7    position 2 incision {{
       var Instrument → desc instrument {{ }}
9    }}
   ]
11 END

13 CONSTRUCT
   incisions [
15   all var Incision
   ]
17 FROM
   in {
19   resource { "file:report1.xml", "xml" },
     desc section [[
21     section.title { "Procedure" },
       desc var Incision → incision {{ }}
23   ]]
   }
25 END
```

The above Xcerpt implementation first builds a list of all incisions (lines 13–25). From this list, the instruments of the second incision can be extracted using Xcerpts **position** construct.

The Xcerpts **position** construct can only be used for querying the position of an element among its siblings (which is the reason why in the above solution it was first necessary to extract the list of all incisions). In XQuery however as illustrated by the solution shown in Listing 2.42, positional expressions (here [2] an abbreviation for the positional predicate [position()=2]) can be applied to any sequence. In Listing 2.42, it is applied to the sequence of all incision descendants of $s as ensured by the brackets around $s//incision. Without the brackets all incision elements under $s that are the second child of their parent would be selected.

Listing 2.42: SEQ-Q1: Implementation in XQuery

```
  for $s in doc("report1.xml")//section[section.title = "Procedure"]
2 return ($s//incision)[2]/instrument
```

Query SEQ-Q2:
In the Procedure section of Report1, what are the first two Instruments to be used?

Listing 2.43: SEQ-Q2: Implementation in Xcerpt

```
  CONSTRUCT
2 result [
    first 2 var Instrument
4 ]
  FROM
6 in {
    resource { "file:report1.xml", "xml" },
8   desc section [[
      section_title { "Procedure" },
10    desc var Instrument → instrument {{}}
    ]]
```

```
12 }
   END
```

This query is even simpler than the previous one, as the result of the positional part of the query is not further queried. Whereas the XQuery solution shown in Listing 2.44 is almost identical to the previous one, the Xcerpt program in Listing 2.43 differs more noticeably, as it is in this case not necessary to first create the list of all instruments, but instead one can simply use the **first** construct as for solving Query XMP-Q6.

Listing 2.44: SEQ-Q2: Implementation in XQuery

```
  for $s in doc("report1.xml")//section[section.title = "Procedure"]
2 return ($s//instrument)[position()<=2]
```

2.4.2 SEQ-Q3,SEQ-Q4,SEQ-Q5: Mixing Absolute and Relative Position

Query SEQ-Q3:
In Report1, what Instruments were used in the first two Actions after the second Incision?

In addition to querying the absolute position as in the previous queries, this query shows how to query the relative position of elements. As for the case of querying absolute positions, there is a noticeable difference in style between the two query languages considered here:

- In Xcerpt querying the relative position of elements is very natural using an ordered term: there, the order of the children or descendants in the data must reflect their order in the query. If one element should follow they other in the data, one uses an ordered term and specifies the element in the appropriate order. This is particularly convenient for selecting data in between other elements (see Query SEQ-Q5).

- XQuery can either use XPath axes such as **following** or **following-sibling** (however, these are optional axes in XQuery) or the >gt; (jbefore') and <lt; (jafter') node comparisons. The latter ones can be used to compare two node w.r.t. their relative document order (for more details see Boag *et al.* (2005)).

Listing 2.45: SEQ-Q3: Implementation in Xcerpt

```
   GOAL
2  result {
     all var Instrument
4  }
   FROM
6  actions {{
     action {{
8      var Instrument → desc instrument {{ }}
     }}
10 }}
   END
12
   CONSTRUCT
14 actions [ first 2 var Action ]
   FROM
16 and {
     incisions [ var Incision → position 2 incision {{ }} ],
18   in {
       resource { "report.xml", "xml" },
```

38

```
20      report [[
          var Incision → desc incision {{ }},
22        var Action → desc action {{ }}
        ]]
24    }
    }
26  END


28

    CONSTRUCT
30  incisions [ all var Incision ],
    FROM
32  in {
      resource { "report.xml", "xml" },
34    var Incision → desc incision {{ }}
    }
36  END
```

To solve this query in Xcerpt (see Listing 2.45) three steps are used: First (lines 29–36) extracts all incision elements from the report. Then the first two actions following the second incision are selected in lines 13–26. Finally all instruments in these action elements are selected (lines 1– 11). Notice the use of an ordered term specification in line 20: thereby only actions after the second incision (bound to the variable Incision) are bound to Action.

The XQuery solution is far more compact, as absolute positions of elements in arbitrary sequences can be directly queried in XQuery. As discussed the after operator is used in line 2 to select only actions after the second incision (selected by $i2).

Listing 2.46: SEQ-Q3: Implementation in XQuery

```
1  let $i2 := (doc("report1.xml")//incision)[2]
   for $a in (doc("report1.xml")//action)[. >> $i2][position()<=2]
3  return $a//instrument
```

Query SEQ-Q4:
In `report1.xml`, find "Procedure" sections where no anesthesia element occurs before the first Incision.

For this and the following query, the Xcerpt solutions demonstrate the convenience of ordered term specifications: instead of using the **position** on appropriately constructed terms, they make use of ordered term specifications and the **without** construct.

Listing 2.47 shows how to implement query SEQ-Q4 in Xcerpt. All sections with title "Procedure" (line 8) and an incision descendant that is *not* preceded by any anesthesia or incision at any depth. In other words: it is the first incision and no anesthesia occurs before it, just as asked by the query.

Listing 2.47: SEQ-Q4: Implementation in Xcerpt

```
1  GOAL
   result [
3    all var Section
   ]
5  FROM
   in {
7    resource { "report1.xml", "xml" },
     desc var Section → section {{
9      section.title { "Procedure" },
       section.content [[
11       without desc anesthesia {{}},
         without desc incision {{}},
13       desc incision {{}}
```

39

```
       ]]
15   }}
   }
17 END
```

The XQuery solution is slightly closer to the formulation of Query SEQ-Q4 but needs to use a fairly complex condition on the section expressed in the **where** clause lines 2–3. It states that the selected `section` may *not* contain any anesthesia that precedes the first incision.

Listing 2.48: SEQ-Q4: Implementation in XQuery

```
  for $p in doc("report1.xml")//section[section.title = "Procedure"]
2 where not(some $a in $p//anesthesia satisfies
         $a << ($p//incision)[1] )
4 return $p
```

The final query in this section is similar, just that in this case everything between two elements is to be selected.

Query SEQ-Q5:
In report1.xml, what happened between the first Incision and the second Incision?

The Xcerpt solution shown in the following Listing is very similar to the one for query XMP-Q4. It collects everything between the first incision (i.e., the incision that is not preceded by another incision) and the next incision. It is insured that it is indeed the next incision by binding only elements with label differing from incision to the variable `Between` and by demanding that there is no other incision element between the two incisions.

Listing 2.49: SEQ-Q5: Implementation in Xcerpt

```
  GOAL
2 result [
    all var Between
4 ]
  FROM
6 in {
    resource { "report1.xml", "xml" },
8   report [[
      without desc incision{{}},
10    desc incision {{}},
      var Between → desc !/incision/ {{ }},
12    without desc incision {{ }},
      desc incision {{}}
14  ]]
  }
16 END
```

The XQuery solution is straightforward with exception of line 4. In line 3 one the result is surprisingly restricted to only the first section with a title "Procedure", something demanded by query XMP-Q4 but not in this case. Regardless, lines 4 and 5 do select the first and section incision in that section. Line 6 is used to avoid descendants of the first incision (as these are in document order after the incision and thereby not rejected by line 7). Finally line 7 ensures that only nodes are returned that are between the two incisions selected in line 4 and 5.

Listing 2.50: SEQ-Q5: Implementation in XQuery

```
1 <critical_sequence>
  {
3  let $proc := doc("report1.xml")//section[section.title="Procedure"][1],
       $i1 :=  ($proc//incision)[1],
5      $i2 :=  ($proc//incision)[2]
```

```
   for $n in $proc//node() except $i1//node()
 7  where $n >> $i1 and $n << $i2
   return $n
 9  }
   </critical_sequence>
```

As pointed out in Chamberlin *et al.* (2005), both solutions might not be considered entirely satisfactory as the result contains duplicates for all nodes that are descendant of another node that is also between the two incisions. Such a node is included twice in the result, once as part of its ancestor, once as result on its own.

Listing 2.51: SEQ-Q5: Duplicate-free Implementation in XQuery

```
   declare function local:between($seq as node()*, $start as node(), $end as node())
 2  as item()*
   {
 4  let $nodes :=
      for $n in $seq except $start//node()
 6    where $n >> $start and $n << $end
      return $n
 8  return $nodes except $nodes//node()
   };
10
   <critical_sequence>
12  {
   let $proc := doc("report1.xml")//section[section.title="Procedure"][1],
14     $first :=  ($proc//incision)[1],
       $second:=  ($proc//incision)[2]
16  return local:between($proc//node(), $first, $second)
   }
18 </critical_sequence>
```

Chamberlin *et al.* (2005) shows a solution for this problem in XQuery that makes use of XQuery's ability to compute the difference of two sequences of nodes using the **except** keyword. A similar outcome can be obtained in Xcerpt as shown in Listing 2.52. That solution uses a rule to construct the same result as above and then selects from that result only those elements for which there are no elements in the result that are their ancestors.

Listing 2.52: SEQ-Q5: Duplicate-free Implementation in Xcerpt

```
 1 GOAL
   duplicate-free-result [
 3   all var X
   ]
 5 FROM
   and {
 7   result [[ var X ]],
     not {
 9     and {
         result [[ var Y ]],
11       in {
           resource { "report1.xml", "xml" },
13         var Y → desc /.*/ {{
             var X → desc /.*/
15         }}
         } }
17   }
   }
19 END

21

   CONSTRUCT
23 result [
     all var Between
25 ]
   FROM
```

41

```
27  in {
       resource { "report1.xml", "xml" },
29     report [[
         without desc incision{{}},
31       desc incision {{}},
         var Between → desc !/incision/ {{ }},
33       without desc incision {{ }},
         desc incision {{}}
35     ]]
     }
37  END
```

2.5 Querying Flat Structures: Use Case "R--Access to Relational Data"

In this section, a scenario for querying data from a relational database but represented in (very regular, flat) XML is presented. In particular, relations between different information items (e.g., users and their bids) are not represented using the XML hierarchy but rather using keys (i.e., unique values of distinguished elements) for identifying and relating information items. There are good reasons for considering such a representation:

- XML is tailored to representing tree-shape (hierarchical) data. When considering relational data that is not tree-shaped, but rather represents an arbitrary graph, not all relations can be represented in the hierarchical (e.g., parent-child, sibling) relations provided by XML. Instead one uses, as in relational databases, unique identifiers (or jkeys') for linking data items. Several such linking mechanisms have been proposed and often standardized, e.g., ID/IDREF links, XLink, or RDF/XML's resource identifieres.

- It is often necessary to integrate access to relational data with access to data in XML format. Also, in practice, a number of XML formats exhibits a close resemblance to the kind of flat, relational data considered in the following use cases.

The queries in this part of Chamberlin *et al.* (2005) are rather similar among each other. Therefore we focus on a few select ones that show features of Xcerpt or XQuery not discussed in the previous sections and can stand as representatives for the omitted queries.

The use case is centered around an auction site, where three relational tables are used for data about auctioned items, users and auction bids. The XML representation of an excerpt of the data is shown in Listing 2.53, for the full data see Chamberlin *et al.* (2005).

Listing 2.53: Excerpt of the XML representation of a relational auction database (referred to as `items.xml`, `users.xml`, `bids.xml`)

```
    <items>
2     <item_tuple>
        <itemno>1001</itemno>
4       <description>Red Bicycle</description>
        <offered_by>U01</offered_by>
6       <start_date>1999-01-05</start_date>
        <end_date>1999-01-20</end_date>
8       <reserve_price>40</reserve_price>
      </item_tuple>
10    <!-- !!! Snip !!! -->

12  <users>
      <user_tuple>
```

```
14      <userid>U01</userid>
        <name>Tom Jones</name>
16      <rating>B</rating>
     </user_tuple>
18  <!-- !!! Snip !!! -->

20  <bids>
     <bid_tuple>
22      <userid>U02</userid>
        <itemno>1001</itemno>
24      <bid>35</bid>
        <bid_date>1999-01-07</bid_date>
26      </bid_tuple>
     <bid_tuple>
28  <!-- !!! Snip !!! -->
```

Observe how the relations are expressed using keys (e.g, in userid and itemno). Also the data is graph shaped but acyclic as, e.g., one item or user can partake in multiple bids, but relations are directed only fom bids to users and items and from items to users but not back from users to bids.

Whereas XQuery uses a tree-shaped data model and therefore can not represent this graph shaped relations using parent/child, Xcerpt's data model is (as, e.g., OEM) graph shaped. The following Xcerpt program queries the data from Listing 2.53 and constructs from it a graph such that each bid contains a user and a item element instead of references to the ID's of the bidding user and item bidded for. Analogously, each item contains a user element nested inside a offered_by for each user offering the item, cf. Figure 2.1.

Listing 2.54: Graph View on Relational Data in Xcerpt

```
1  CONSTRUCT
   auction-graph {
3    var Bids,
     var Items,
5    var Users
   }
7  FROM
   and {
9    var Bids → bids {{ }},
     var Items → items {{ }},
11   var Users → users {{ }},
   }
13 END

15 CONSTRUCT
   bids {
17   all bid {
       ^var UserID,
19       ^var ItemNo,
       all var OtherBidAttributes
21   } group by { var Bid }
   }
23 FROM
   and {
25   in {
       resource { "file:bids.xml", "xml" },
27     bids {{
         var Bid → bid_tuple {{
29         userid [ var UserID ],
           itemno [ var ItemNo ],
31         var OtherBidAttributes
         }}
33     }}
     },
35
   }
```

```
37  END

39  CONSTRUCT
    items {
41    all var ItemNo@item {
        itemno [ var ItemNo ],
43      offered_by [ ^varUserID ],
        all var Attributes
45    }
    }
47  FROM
    in {
49    resource { "file:items.xml", "xml" },
      items {{
51      item_tuple {{
          itemno [ var ItemNo ],
53        offered_by [ var UserID ],
          var Attributes
55      }}
      }}
57  }
    END
59
    CONSTRUCT
61  users {
      all var UserID@user {
63      userid {{ var UserID }},
        all var Attributes
65    }
    }
67  FROM
    in {
69    resource { "file:users.xml", "xml" },
      users {{
71      user_tuple {{
          userid {{ var UserID }},
73        var Attributes
        }}
75    }}
    }
77  END
```

Queries on such a graph view can avoid explicit value-based joins relating information through the structure of the XML data. However, choosing a proper graph view is not always obvious: E.g., the links in relational data are undirected. whereas in the above graph view the relations are directed from bids to items and users. For most of the below queries a graph view where the bids are reachable from the items and users is indeed preferable to the one shown above. E.g., query R-Q2 asks for all items together with their bids if they have any. On the above graph view this query requires special handling of items without bids (see Listing 2.59).

2.5.1   R-Q1: Value-based selection and ordering on relational data

Query R-Q1:
List the item number and description of all bicycles that currently have an auction in progress, ordered by item number.

The first query operates only on data on items (therefore also does not benefit from the graph view). For selecting only items that are currently being auctioned a comparison of the values of an item's start_date and end_date against the current date is required. Such selection based on the value rather than on the structure (or relations) of data is typical for flat, relational data. Value-based comparisons depend on the type of the (scalar) data (or more precisely, on
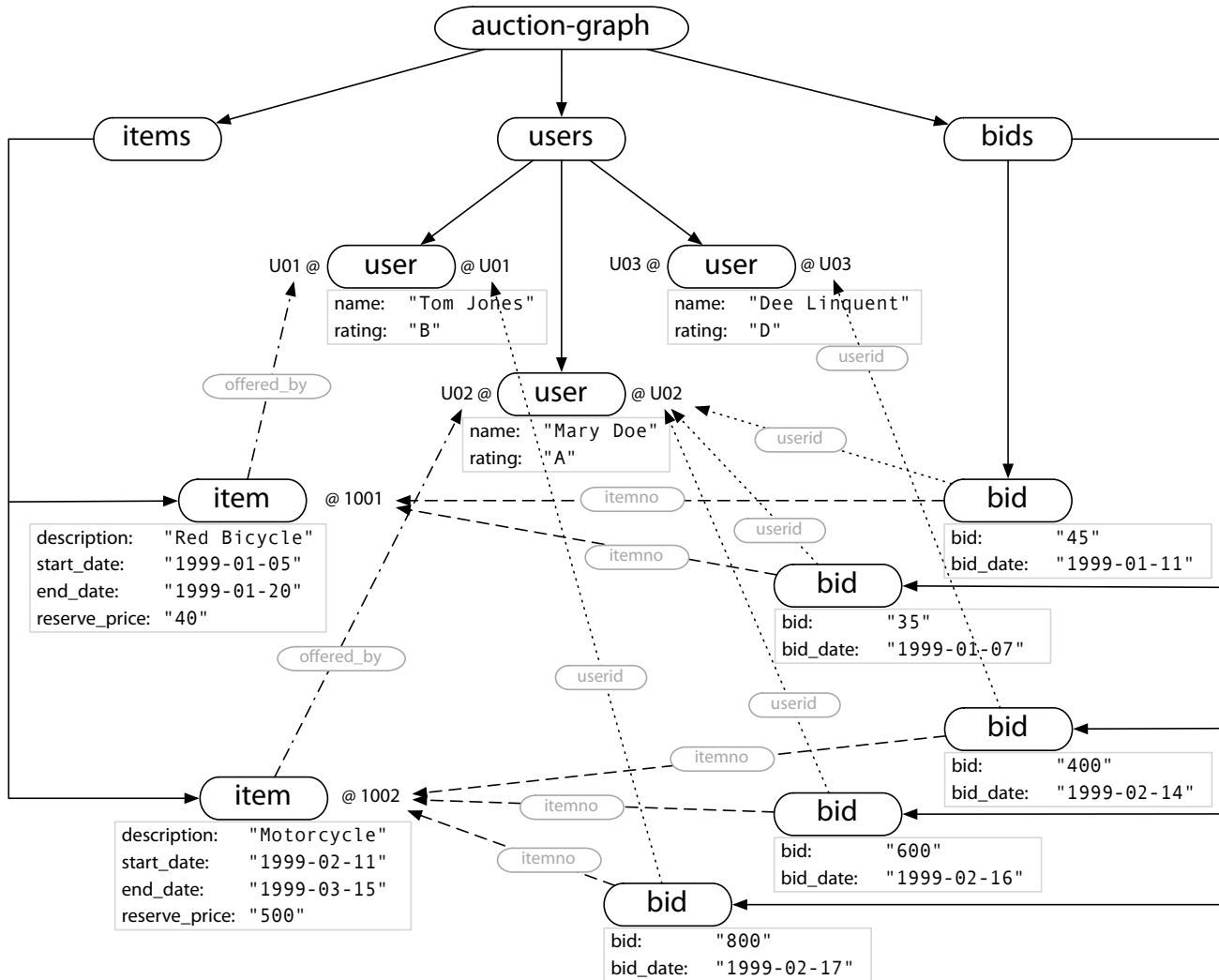
44

Figure 2.1: Illustration of auction-graph View

the order relation on the data, e.g., successor order on integers vs. lexical order on strings). XQuery uses type information (obtained, from a schema or explicitly stated using type casts) to determine what comparison method to use (see Malhotra *et al.* (2005) for comparison operators on XQuery's scalar (or "simple") types). Xcerpt's type system is still under development but will use a similar mix of implicit and explicit type information for selecting a proper comparison method. The same considerations apply for ordering constructs such as the **order by** clause in XQuery or Xcerpt. Additionally, Xcerpt's **order by** admits an explicit statement of the method for ordering (e.g., **numeric** or **lexical**).

Listing 2.55: R-Q1: Implementation in Xcerpt

```
   GOAL
 2 result [
     all item_tupel {
 4     itemno {
         var Itemno
 6     },
       all var Description
 8   } order by [var Itemno] numeric
   ]
10 FROM
   in {
12   resource { "file:items.xml", "xml" },
     items {{
14     item_tuple {{
         itemno {{ var Itemno }},
16       var Description → description {{
           /.*Bicycle.*/
18       }},
         start_date { var Startdate },
20       end_date { var Enddate }
       }}
22   }}
   }
24 where {
     and {
26     var Startdate <= current-date(),
       var Enddate >= current-date()
28   }
   }
30 END
```

Both solutions for Query R-Q1 use **order by** clauses for sorting the result and value-based comparisons for properly restricting the selected items. Observe, how the Xcerpt solution separates such additional restrictions into a **where** clause at the end of the query term. XQuery follows a different philosophy by keeping the restrictions close to the variable definition (i.e., **for** clause). This can be mimicked in Xcerpt by replacing the one **where** clause at the end of the query term with two where clauses in line XX and XXX, the first restricting the start, the second restriction the end date.

Listing 2.56: R-Q1: Implementation in XQuery

```
 1 <result>
     {
 3     for $i in doc("items.xml")//item_tuple
       where $i/start_date <= current-date()
 5       and $i/end_date >= current-date()
         and contains($i/description, "Bicycle")
 7     order by $i/itemno
       return
 9         <item_tuple>
               { $i/itemno }
11             { $i/description }
```

```
       </item_tuple>
13   }
   </result>
```

2.5.2   R-Q2: Value-based Joins and Aggregation

In addition to value-based selection and constructions, the other major aspect of the queries in this section are value-based joins and aggregation. Value-based joins allow to query the relations in the data not expressed in the XML structure. As shown in the introduction of this section, such non-structural relations can be explicitly expressed in the structure, however the resulting structures are in general *not* tree shaped (and therefore can not be constructed in XQuery).

Query R-Q2:
For all bicycles, list the item number, description, and highest bid (if any), ordered by item number.

On the original XML data, the Xcerpt and the XQuery solution for this query are very straightforward. In both cases, matching items are selected together with bids that have the same itemno.

Listing 2.57: R-Q2: Implementation in Xcerpt

```
1  GOAL
   result [
3    all item_tupel {
       itemno {
5        var Itemno
       },
7      var Description,
       optional high_bid {
9        max ( all var Bid )
       }
11   } order by [ var Itemno ] numeric
   ]
13 FROM
   and {
15   in {
       resource { "file:items.xml", "xml" },
17     items {{
         item_tuple {{
19         itemno {{ var Itemno }},
           var Description → description {{ /.*Bicycle.*/ }}
21       }}
       }}
23   },
     optional in {
25     resource { "file:bids.xml", "xml" },
       bids {{
27       bid_tuple {{
           itemno {{ var Itemno }},
29          bid { var Bid }
         }}
31     }}
     }
33 }
   END
```

Notice, how the Xcerpt solution uses an explicit optional keyword to indicate that the second conjunct in the query term may not be matched by any data, but if there is matching data

47

it is retrieved. In XQuery almost the same effect is obtained by a **let** expression for binding the *sequence* of all bids with the same itemno as the current item. Thus $b can be bound to the empty sequence in the case there are no bids for an item at all.

In fact, there is a subtle difference between the two solutions. Listing 2.57 does not create a high_bid element if there are no bids for an item, whereas the XQuery solution does. It is possible to obtain the prior behavior in XQuery by adding a conditional expression testing whether $b is the empty sequence around the high_bid element constructor. Dually, the latter behavior can be obtained in Xcerpt by moving the **optional** inside the aggregation function.

Listing 2.58: R-Q2: Implementation in XQuery

```
1  <result>
     {
3      for $i in doc("items.xml")//item_tuple
       let $b := doc("bids.xml")//bid_tuple[itemno = $i/itemno]
5      where contains($i/description, "Bicycle")
       order by $i/itemno
7      return
           <item_tuple>
9              { $i/itemno }
               { $i/description }
11             <high_bid>{ max($b/bid) }</high_bid>
           </item_tuple>
13   }
   </result>
```

A solution for this query on the graph view from Listing 2.54 is shown in the following program. Notice how the value-based join on the itemno can be avoided using structure matching instead (the item is nested inside the corresponding bids). However, a disjunction is needed to cover the case of items without bids.

Listing 2.59: R-Q2: Implementation in Xcerpt on auction-graph view

```
1  GOAL
   result [
3    all item_tupel {
       itemno {
5        var Itemno
       },
7      var Description,
       optional high_bid {
9        max ( all var Bid )
       }
11   } order by [ var Itemno ] numeric
   ]
13 FROM
   or {
15   auction-graph {{
       bids {{
17       bid {{
           bid { var Bid },
19         item {{
             itemno {{ var Itemno }},
21           var Description → description {{ /.*Bicycle.*/ }}
           }}
23       }}
       }}
25   }},
     auction-graph {{
27     items {{
         item {{
29         itemno {{ var Itemno }},
           var Description → description {{ /.*Bicycle.*/ }}
31       }}
       }}
```

```
33    }}
     }
35  END
```

2.5.3   R-Q5 to R-Q13: Complex Aggregation and Grouping

Where the previous two queries have focused on illustrating typical functionalities required for querying flat, relational data, the remaining queries and solutions in this section are a representative selection of the increasingly complex aggregation and grouping queries from Section 1.3 of Chamberlin *et al.* (2001) with few comments on constructs or paradigms not observed in previous solutions.

Query R-Q6:
For each item whose highest bid is more than twice its reserve price, list the item number, description, reserve price, and highest bid.

Listing 2.60: R-Q6: Implementation in Xcerpt

```
    CONSTRUCT
2   highest_bids {
      all highest_bid {
4       for_item { var Itemno},
        high_bid {
6         max ( all var Bid )
        }
8     }
    }
10  FROM
    in {
12    resource { "file:bids.xml", "xml" },
      bids {{
14      bid_tuple {{
          itemno { var Itemno },
16        bid { var Bid }
        }}
18    }}
    }
20  END

22  GOAL
    result {
24    all successful_item {
        itemno { var Itemno },
26      description { var Description },
        reserve_price { var ReservePrice },
28      high_bid { var HighestBid }
      }
30  }
    FROM
32  and {
      in {
34      resource { "file:items.xml", "xml" },
        items {{
36        item_tuple {{
            itemno { var Itemno },
38          description { var Description },
            reserve_price { var ReservePrice }
40        }}
        }}
42    },
      highest_bids {{
44      highest_bid {{
          for_item { var Itemno },
```

```
46      high_bid { var HighestBid }
        }
48    }}
   }}
50 } where { var HighestBid > var ReservePrice * 2 }
   END
```

In the Xcerpt solution, the highest bid for each item is computed in a separate rule. It is not possible to express the query in a single Xcerpt rule as aggregations is considered a form of construction in Xcerpt and can therefore only be computed in a construct term (i.e., only in the head of a rule). As XQuery does not have this strict separation, most aggregation queries can be formulated more compact.

Listing 2.61: R-Q6: Implementation in XQuery

```
<result>
2  {
     for $item in doc("items.xml")//item_tuple
4    let $b := doc("bids.xml")//bid_tuple[itemno = $item/itemno]
     let $z := max($b/bid)
6    where $item/reserve_price * 2 < $z
     return
8        <successful_item>
             { $item/itemno }
10           { $item/description }
             { $item/reserve_price }
12           <high_bid>{$z }</high_bid>
          </successful_item>
14   }
   </result>
```

Query R-Q10:
For each item that has received a bid, list the item number, the highest bid, and the name of the highest bidder, ordered by item number.

Listing 2.62: R-Q10: Implementation in Xcerpt

```
   CONSTRUCT
2  highest_bids ... (from R-Q6)
   END

4
   GOAL
6  result [
     all high_bid [
8      itemno { var Itemno },
       bid { var HighestBid },
10     all bidder { var Name }
     ] order by [ var Itemno ] numeric
12 ]
   FROM
14 and {
     in {
16     resource { "file:bids.xml", "xml" },
       bids {{
18       bid_tuple {{
           itemno { var Itemno },
20         bid { var HighestBid },
           userid { var Userid }
22       }}
       }}
24   },
     in {
26     resource { "file:users.xml", "xml" },
       users {{
```

```
28      user_tuple {{
          userid { var Userid },
30        name { var Name }
        }}
32    }}
    },
34  highest_bids {{
      highest_bid {{
36      for_item { var Itemno },
        high_bid { var HighestBid }
38      }
      }}
40  }}
  }
42 END
```

The Xcerpt solution uses the same rule as in Listing 2.60 to compute the highest bidding for an item. Based on the result of that rule all the highest bids are selected. Depending on the auctioning system there might be several such bids, thus several highest bidders collected using the **all** keyword in line 10.

The XQuery solution is unremarkable, but does not consider multiple highest bids. As seen above, such grouping would require a nested query in line 12.

Listing 2.63: R-Q10: Implementation in XQuery

```
1 <result>
   {
3     for $highbid in doc("bids.xml")//bid_tuple,
          $user in doc("users.xml")//user_tuple
5     where $user/userid = $highbid/userid
        and $highbid/bid = max(doc("bids.xml")//bid_tuple[itemno=$highbid/itemno]/bid)
7     order by $highbid/itemno
      return
9         <high_bid>
              { $highbid/itemno }
11            { $highbid/bid }
              <bidder>{ $user/name/text() }</bidder>
13        </high_bid>
   }
15 </result>
```

Query R-Q11:
List the item number and description of the item(s) that received the highest bid ever recorded, and the amount of that bid.

Listing 2.64: R-Q11: Implementation in Xcerpt

```
  CONSTRUCT
2 highest_bid_ever {
    max ( var Bid )
4 }
  FROM
6 in {
    resource { "file:bids.xml", "xml" },
8   bids {{
      bid_tuple {{
10      itemno { var Itemno },
        bid { var Bid }
12      }}
    }}
14 }
  END

16
```

```
   GOAL
18 result {
     all expensive_item {
20     itemno { var Itemno },
       bid { var HighestBidEver },
22     description { var Description }
     }
24 }
   FROM
26 and {
     highest_bid_ever { var HighestBidEver }
28   in {
       resource { "file:items.xml", "xml" },
30     items {{
         item_tuple {{
32         itemno { var Itemno },
           description { var Description }
34       }}
       }}
36   },
     in {
38     resource { "file:bids.xml", "xml" },
       bids {{
40       bid_tuple {{
           itemno { var Itemno },
42         bid { var HighestBidEver }
         }}
44     }}
     }
46 }
   END
```

Again a separate rule is needed in the Xcerpt query to compute the highest bid ever. Given that information the remainder of the query is straightforward, as the XQuery solution.

Listing 2.65: R-Q11: Implementation in XQuery

```
   let $highbid := max(doc("bids.xml")//bid_tuple/bid)
2  return
       <result>
4      {
           for $item in doc("items.xml")//item_tuple,
6            $b in doc("bids.xml")//bid_tuple[itemno = $item/itemno]
           where $b/bid = $highbid
8          return
               <expensive_item>
10                 { $item/itemno }
                   { $item/description }
12                 <high_bid>{ $highbid }</high_bid>
               </expensive_item>
14     }
       </result>
```

Query R-Q12:

List the item number and description of the item(s) that received the largest number of bids, and the number of bids it (or they) received.

Listing 2.66: R-Q12: Implementation in Xcerpt

```
   CONSTRUCT
2  items_with_bid_counts {
     all item_popularity {
4      for_item { var Itemno},
       bid_count {
6        count ( all var Bid )
```

```
          }
 8     }
     }
10  FROM
     in {
12     resource { "file:bids.xml", "xml" },
       bids {{
14       bid_tuple {{
           itemno { var Itemno },
16         bid { var Bid }
         }}
18     }}
     }
20  END

22  CONSTRUCT
     highest_bid_count {
24     max ( var BidCount )
     }
26  FROM
     items_with_bid_counts {{
28     item_popularity {{ bid_count { var BidCount } }}
     }}
30  END

32  GOAL
     result {
34     all popular_item {
         itemno { var Itemno },
36       var Description,
         var BidCount
38     }
     }
40  FROM
     and {
42     highest_bid_count {
         var HighestBidCount
44     },
       items_with_bid_counts {{
46       item_popularity {{
           for_item { var Itemno },
48         var BidCount → bid_count { var HighestBidCount }
         }}
50     }},
       in {
52       resource { "file:items.xml", "xml" },
         items {{
54         item_tuple {{
             itemno { var Itemno },
56           var Description → description {{ }}
           }}
58       }}
       }
60  }
     END
```

In this case the two solutions are more similar than in the previous cases, as also the XQuery solution constructs separately a list of items with the number of their bids. Observe, how the result of the user defined functiuon `local:bid_summary()` is further queried in the main query part of the XQuery solution. The Xcerpt solution is basically an amalgamation of the solutions for R-Q11 and R-Q10 with number of bids instead of value of bids as aggregated values.

Listing 2.67: R-Q12: Implementation in XQuery

```
declare function local:bid_summary()
2   as element()*
```

```
   {
4      for $i in distinct-values(doc("bids.xml")//itemno)
       let $b := doc("bids.xml")//bid_tuple[itemno = $i]
6      return
           <bid_count>
8              <itemno>{ $i }</itemno>
               <nbids>{ count($b) }</nbids>
10         </bid_count>
   };
12
   <result>
14 {
       let $bid_counts := local:bid_summary(),
16         $maxbids := max($bid_counts/nbids),
           $maxitemnos := $bid_counts[nbids = $maxbids]
18     for $item in doc("items.xml")//item_tuple,
           $bc in $bid_counts
20     where $bc/nbids = $maxbids and $item/itemno = $bc/itemno
       return
22         <popular_item>
               { $item/itemno }
24             { $item/description }
               <bid_count>{ $bc/nbids/text() }</bid_count>
26         </popular_item>
   }
28 </result>
```

2.5.4 R-Q14,R-Q18: Ordering

The remaining two queries in this part cover again the topic of ordering, but in a slightly more challenging context: R-Q14 requires ordering on an aggregated value, R-Q18 shows that ordered result is often nested, requiring ordering on nested groupings to be constructed.

Query R-Q14:
List item numbers and average bids for items that have received three or more bids, in descending order by average bid.

Listing 2.68: R-Q14: Implementation in Xcerpt

```
1 CONSTRUCT
  items_with_bidcount_avgbid {
3   all item {
      itemno { var Itemno  },
5     bidcount {
        count ( all var Bid )
7     },
      avgbid {
9       avg ( all var Bid )
      }
11  }
  }
13 FROM
  in {
15  resource { "file:bids.xml", "xml" },
    bids {{
17    bid_tuple {{
        itemno { var Itemno },
19      bid { var Bid }
      }}
21  }}
  }
23 END

25 GOAL
```

```
   result [
27   all popular_item {
       itemno { var Itemno },
29     avgbid { var Avgbid }
     } order by [ var Avgbid ] numeric descending
31 ]
   FROM
33 items_with_bidcount_avgbid {{
     item {{
35     itemno { var Itemno },
       bidcount { var BidCount },
37     avgbid { var Avgbid }
     }}
39 }}
   where { var Bidcount >= 3 }
41 END
```

As expected, the Xcerpt solution requires a separate rule to compute the two aggregations. Aside of that both solutions are straightforward using the **descending** to indicate the sort direction.

Listing 2.69: R-Q14: Implementation in XQuery

```
   <result>
2  {
     for $i in distinct-values(doc("bids.xml")//itemno)
4    let $b := doc("bids.xml")//bid_tuple[itemno = $i]
     let $avgbid := avg($b/bid)
6    where count($b) >= 3
     order by $avgbid descending
8    return
         <popular_item>
10          <itemno>{ $i }</itemno>
            <avgbid>{ $avgbid }</avgbid>
12       </popular_item>
   }
14 </result>
```

Query R-Q18:

List all users in alphabetic order by name. For each user, include descriptions of all the items (if any) that were bid on by that user, in alphabetic order.

Listing 2.70: R-Q18: Implementation in Xcerpt

```
1  GOAL
   result {
3    all user [
       name { var Name },
5      optional all bid_on_item {
         var Description
7      } order by [ var Description ] lexical
     ] order by [ var Name ] lexical
9  }
   FROM
11 and {
     in {
13     resource { "file:users.xml", "xml" },
       users {{
15       user_tuple {{
           userid { var Userid },
17         name { var Name }
         }}
19     }}
     },
21   optional in {
```

```
          resource { "file:bids.xml", "xml" },
23        bids {{
            bid_tuple {{
25            userid { var Userid },
              itemno { var Itemno }
27          }}
          }}
29      }
      optional in {
31      resource { "file:items.xml", "xml" },
          items {{
33          item_tuple {{
              itemno { var Itemno },
35            description { var Description }
            }}
37      }}
        },
39 }
    END
```

Xcerpt's solution performs the necessary joins in the query term and returns all users and, if there are any such items, also the items a user is bidding on. In contrast to the XQuery case, Xcerpt's grouping construct **all** includes a duplicate removal. In the XQuery solution nested queries must be used for grouping (as discussed in detail above). In line 9 **distinct-values** ensures that items where a single user has several bids on are returned once only.

Listing 2.71: R-Q18: Implementation in XQuery

```
1 <result>
    {
3     for $u in doc("users.xml")//user_tuple
      order by $u/name
5     return
          <user>
7             { $u/name }
              {
9                 for $b in distinct-values(doc("bids.xml")//bid_tuple
                                            [userid = $u/userid]/itemno)
11                for $i in doc("items.xml")//item_tuple[itemno = $b]
                  let $descr := $i/description/text()
13                order by $descr
                  return
15                    <bid_on_item>{ $descr }</bid_on_item>
              }
17        </user>
    }
19 </result>
```

2.6 Querying Rich and Recursive Structures: Use Case "SGML-- Standard Generalized Markup Language"

As a contrast to the previous section, the following data and queries are richly structured data typical for what has become known as document-oriented XML: The schema is very flexible, usually recursive in depth. Therefore queries make ample use of "incomplete" query constructs where the shape of the structure is only incompletely specified in the query, e.g., the **desc** in Xcerpt or **descendant** in XQuery for traversing arbitrary length paths.

Listing 2.72: Excerpt of an XML report (referred to as `sgml.xml`)

```
<!DOCTYPE report SYSTEM "report.dtd">
```

```
 2 <report>
   <title>Getting started with SGML</title>
 4 <chapter>
    <title>The business challenge</title>
 6 <intro>
     <para>With the ever-changing and growing global market, companies and
 8     large organizations are searching for ways to become more viable and
       competitive. Downsizing and other cost-cutting measures demand more
10     efficient use of corporate resources. One very important resource is
       an organization\char39 s information.</para>
12   <para>As part of the move toward integrated information management,
       whole industries are developing and implementing standards for
14     exchanging technical information. This report describes how one such
       standard, the Standard Generalized Markup Language (SGML), works as
16     part of an overall information management strategy.</para>
     <graphic graphname="infoflow"/></intro></chapter>
18 <chapter>
    <title>Getting to know SGML</title>
20   <intro>
     <para>While SGML is a fairly recent technology, the use of
22     <emph>markup</emph> in computer-generated documents has existed for a
       while.</para></intro>
24   <section shorttitle="What is markup?">
     <title>What is markup, or everything you always wanted to know about
26     document preparation but were afraid to ask?</title>
     <intro>
28     <para>Markup is everything in a document that is not content. The
         traditional meaning of markup is the manual <emph>marking</emph> up
30       of typewritten text to give instructions for a typesetter or
         compositor about how to fit the text on a page and what typefaces to
32       use. This kind of markup is known as <emph>procedural markup</emph>.
        </para>
34     </intro>
     <topic topicid="top1">
36     <title>Procedural markup</title>
       <para>Most electronic publishing systems today use some form of
38       procedural markup. Procedural markup codes are good for one
         presentation of the information.</para></topic>
40   <topic topicid="top2">
       <title>Generic markup</title>
42     <para>Generic markup (also known as descriptive markup) describes the
         <emph>purpose</emph> of the text in a document. A basic concept of
44       generic markup is that the content of a document must be separate from
         the style. Generic markup allows for multiple presentations of the
46       information.</para></topic>
     <topic topicid="top3">
48     <title>Drawbacks of procedural markup</title>
       <para>Industries involved in technical documentation increasingly
50       prefer generic over procedural markup schemes. When a company changes
         software or hardware systems, enormous data translation tasks arise,
52       often resulting in errors.</para></topic></section>
   <section shorttitle="What is SGML?">
54   <title>What <emph>is</emph> SGML in the grand scheme of the universe, anyway?
     </title>
56   <intro>
       <para>SGML defines a strict markup scheme with a syntax for defining
58       document data elements and an overall framework for marking up
         documents.</para>
60   [...]
   </chapter>
62 <chapter>
    <title>Resources</title>
64   <section>
     <title>Conferences, tutorials, and training</title>
66   <intro>
       <para>The Graphic Communications Association has been
68       instrumental in the development of SGML. GCA provides conferences,
         tutorials, newsletters, and publication sales for both members and
70       non-members.</para>
       <para security="c">Exiled members of the former Soviet Union\char39 s secret
```

```
72          police, the KGB, have infiltrated the upper ranks of the GCA and are
            planning the Final Revolution as soon as DSSSL is completed.</para>
74      </intro>
      </section>
76  </chapter>
  </report>
```

The schema of the data (shown in Chamberlin *et al.* (2001)) is indeed recursive (e.g., emph elements may contain other emph elements. Also several elements (e.g., title) may occur at different depths and inside parent elements with different tags.

2.6.1 SGML-Q1 to SGML-Q4: Arbitrary Depth Selection

Query SGML-Q1:
Locate all paragraphs in the report (all para elements occurring anywhere within the report element).

Listing 2.73: SGML-Q1: Implementation in Xcerpt

```
  GOAL
2 results [
    all var Para
4 ]
  FROM
6 in {
    resource { "file:sgml.xml" },
8   desc report [[
      desc var Para → para {{ }}
10  ]]
  }
12 END
```

This basic selection query highlights how to find documents at unknown depth (actually, only the para elements may occur at different depths, but for better comparison with the XQuery solution from Chamberlin *et al.* (2001) also the report elements are searched at any depth in the document). The two main differences that can be observed between the Xcerpt and the XQuery solution are the separation of querying and construction in Xcerpt and the use of patterns in Xcerpt vs. path expressions in XQuery. Note the use of square brackets in Listing 2.73 to ensure that the paragraphs are returned in document order.

Listing 2.74: SGML-Q1: Implementation in XQuery

```
1 <result>
    {
3     doc("sgml.xml")//report//para
    }
5 </result>
```

It is, of course, also possible to solve this query using XQuery's FLWOR expressions, see Listing 2.75. However, in contrast to path expressions where the result is duplicate free, FLWOR expressions may create duplicates. E.g., if a paragraph is nested inside a report that in turn is part of another report that paragraph will be returned twice by the solution in Listing 2.75. This case has to be treated. e.g., by adding **distinct-values** to filter duplicates from the result sequence of the FLWOR expression.

Listing 2.75: SGML-Q1: Implementation in XQuery using FLWOR expressions

```
1 <result>
```

```
      {
3      for $report in doc("sgml.xml")//report
       for $para in $report//para
5      return $para
      }
7 </result>
```

Query SGML-Q2:

Locate all paragraph elements in an introduction (all para elements directly contained within an intro element).

This query varies the theme of the previous one by adding a "directly contained" condition to the query. The resulting solutions differ in dropping the second **desc** or **//** (i.e., **/descendant-or-self::node()/**) expression, thus looking for the para element only directly under a intro element.

Listing 2.76: SGML-Q2: Implementation in Xcerpt

```
1 GOAL
  results [
3    all var Para
  ]
5 FROM
  in {
7    resource { "file:sgml.xml" },
     desc intro [[
9      var Para → para {{}}
     ]]
11 }
   END
```

Listing 2.77: SGML-Q2: Implementation in XQuery

```
1 <result>
   {
3    doc("sgml.xml")//intro/para
   }
5 </result>
```

Query SGML-Q3:

Locate all paragraphs in the introduction of a section that is in a chapter that has no introduction (all para elements directly contained within an intro element directly contained in a section element directly contained in a chapter element. The chapter element must not directly contain an intro element).

Again paragraphs are searched with an increasingly complex condition. The most interesting part of this query is the "has no introduction". Xcerpt's solutions expresses this condition using the **without** keyword that requires the contained sub-query *not* to match.

Listing 2.78: SGML-Q3: Implementation in Xcerpt

```
1 GOAL
  results [
3    all var Para
  ]
5 FROM
  in {
7    resource { "file:sgml.xml" },
     desc chapter {{
```

```
 9    without intro {{}},
      section [[
11      intro [[
          var Para → para {{}}
13      ]]
      ]]
15  }}
  }
17 END
```

In XQuery the same condition can be expressed using any of the tests for an empty sequence, e.g., *empty* or *not* (which converts the sequence to a boolean expressions: an empty sequence converts to false any non-empty sequence to true). As with the above queries, also SGML-Q3 can be expressed using a single path expression inside the result element constructor, see Listing 2.80. Again the main difference between Listings 2.79 and 2.80 is the lack of duplicate removal in the prior one. If a paragraph occurs in two or more (nested) chapters (this is not allowed by the schema therefore ignored in the first solution) it is returned twice.

Listing 2.79: SGML-Q3: Implementation in XQuery

```
  <result>
2   {
      for $c in doc("sgml.xml")//chapter
4     where empty($c/intro)
      return $c/section/intro/para
6   }
  </result>
```

Listing 2.80: SGML-Q3: Implementation in XQuery without FLWOR expressions

```
1 <result>
    {
3     doc("sgml.xml")//chapter[not(intro)]/section/intro/para
    }
5 </result>
```

Query SGML-Q4:
Locate the second paragraph in the third section in the second chapter (the second para element occurring in the third section element occurring in the second chapter element occurring in the report).

Listing 2.81: SGML-Q4: Implementation in Xcerpt

```
1 CONSTRUCT
    chapters [
3     all var Chapter
    ]
5 FROM
    in {
7     resource { "file:report.xml" },
      report [[
9       var Chapter → chapter [[]]
      ]]
11  }
  END
13
  CONSTRUCT
15  sections_of_2nd_chapter [
      all var Section
17  ]
  FROM
```

```
19   chapters [[
        position 2 chapter [[
21        var Section → section [[ ]]
        ]]
23    ]]
   END

25
   CONSTRUCT
27   paragraphs_of_3rd_section_of_2nd_chapter [
        all var Para
29    ]
   FROM
31   sections_of_2nd_chapter [[
        position 3 section [[
33        desc var Para → para {{}}
        ]]
35    ]]
   END

37
   GOAL
39   result [
        var Para
41    ]
   FROM
43   paragraphs_of_3rd_section_of_2nd_chapter [
        position 2 var Para → para {{ }}
45    ]
   END
```

As discussed at length in Section 2.4 querying absolute order within elements of the
same tag requires in Xcerpt to first collect all such elements. Therefore the above query re-
quires three rules that collect all chapters, all sections of the second chapter and all para-
graphs of the third section of that chapter. In XQuery such queries can be expressed far
more succinct using positional predicates (see Listing 2.82). Placing positional predicates
is non-trivial, as, e.g., the expressions `child::a/descendant::b[1]`, `child::a//b[1]`, and
`(child::a//b)[1]` select different elements. The first selects the first b descendant of each
a child, the second selects all b descendants of a children, if they are the first child of their
parent, and the third selects only the very first b descendant of any a child (and is equiva-
lent to `(child::a/descendant::b)[1]`). These rich distinctions, while allowing very succinct
expressions, are often overwhelming for non-experts. For the above query, the third case is
necessary as shown in Listing 2.82.

Listing 2.82: SGML-Q4: Implementation in XQuery

```
1 <result>
   {
3    (((doc("sgml.xml")//chapter)[2]//section)[3]//para)[2]
   }
5 </result>
```

2.6.2   SGML-Q5,SGML-Q6: Querying Attributes

Query SGML-Q5:
*Locate all classified paragraphs (all para elements whose security attribute has the
value "c").*

Another variation on SGML-Q1, this time adding a value-based condition on the security at-
tribute. The formulation is straightforward in both Xcerpt and XQuery. Notice Xcerpt's slightly

61

bulky **attributes** notation for XML attributes. A more concise notation is under consideration, but not yet finalized.

Listing 2.83: SGML-Q5: Implementation in Xcerpt

```
1 GOAL
  results [
3   all var Para
  ]
5 FROM
  in {
7   resource { "file:sgml.xml" },
    desc var Para → para [[
9     attributes  {{
        security [ "c" ]
11    }}
    ]]
13 }
  END
```

XQuery uses `@security` as a concise, if slightly obscure, abbreviation for `attribute::security`.

Listing 2.84: SGML-Q5: Implementation in XQuery

```
1 <result>
    {
3     doc("sgml.xml")//para[@security = "c"]
    }
5 </result>
```

Query SGML-Q6:
List the short titles of all sections (the values of the shorttitle attributes of all section elements, expressing each short title as the value of a new element.)

Essentially this query asks for the creation of an XML element based upon the value of an attribute. No surprises in either of the two solutions.

Listing 2.85: SGML-Q6: Implementation in Xcerpt

```
1 GOAL
    results [
3     all stitle [ var STitle ]
    ]
5 FROM
    in {
7     resource { "file:sgml.xml" },
      desc section [[
9       attributes  {{
          shorttitle [ var STitle ]
11      }}
      ]]
13   }
  END
```

Listing 2.86: SGML-Q6: Implementation in XQuery

```
1 <result>
    {
3     for $s in doc("sgml.xml")//section/@shorttitle
      return <stitle>{ $s }</stitle>
5   }
  </result>
```

The XQuery solution demonstrates once more the subtle differences between FLWOR and path expressions regarding duplicate handling: Once might consider Listing 2.86 and 2.87 equivalent. But the first creates only one stitle element even if two sections have shorttitle atributes with the same value. The latter creates duplicates in the result in that case.

Listing 2.87: SGML-Q6: Implementation in XQuery without Duplicate Removal

```
1 <result>
  {
3   for $s in doc("sgml.xml")//section
    return <stitle>{ $s/@shorttitles }</stitle>
5 }
  </result>
```

2.6.3 SGML-Q7,SGML-Q8a,SGML-Q8b: Complex Content Queries

Query SGML-Q7:
Locate the initial letter of the initial paragraph of all introductions (the first character in the content [character content as well as element content] of the first para element contained in an intro element).

The following three queries show more complex conditions on the value or content of elements. Notice, however, that neither XQuery nor Xcerpt provide convenient full-text querying (cf. Buxton and Rys (2003)) yet. There is an effort to create a full-text extension for XQuery Amer-Yahia *et al.* (2005).

Listing 2.88: SGML-Q7: Implementation in Xcerpt

```
1 CONSTRUCT
  first_paragraphs [
3   all [ first ( all var Para ) ] group by { var Intro }
  ]
5 FROM
  in {
7   resource { "file:sgml.xml" },
    var Intro → desc intro [[
9     var Para → para [[]]
    ]]
11 }
  END
13
  GOAL
15 result [
    all first_letter {
17     first ( all var FirstLetterOfContainedTextNode )
    } group by { var Para }
19 ]
  FROM
21 first_paragraphs [[
    var Para → para [[
23     desc /(var FirstLetterOfContainedTextNode → .).*/
    ]]
25 ]]
  END
```

In the Xcerpt solution, a list of all first paragraphs of introductions is created. From this list the first letter of all contained text nodes is selected in the variable `FirstLetterOfContainedTextNode`. From all these letters of each paragraph only the first is returned in the construct term.

The XQuery solution differs here notably: First it can use positional predicates to select the first paragraph of each introduction. Second XQuery has the concept of the "string value" of a node, that is the concatenation of all text nodes contained in a node. This string value of each paragraph is computed by `string($i)` and the first letter is obtained using the **substring** function. In constrast to the Xcerpt solution it creates duplicates if several paragraphs start with the same letter. Avoiding duplicates is in this case non-trivial, see Section 2.4.

Listing 2.89: SGML-Q7: Implementation in XQuery

```
1 <result>
    {
3     for $i in doc("sgml.xml")//intro/para[1]
      return
5         <first_letter>{ substring(string($i), 1, 1) }</first_letter>
    }
7 </result>
```

Query SGML-Q8a:
Locate all sections with a title that has "is SGML" in it. The string may occur anywhere in the descendants of the title element, and markup boundaries are ignored.

This query and query SGML-Q8b demonstrate how well a query language can query text limited or stretching beyond markup boundaries. Indeed, for document-oriented XML such as XHTML this is an important class of queries.

Listing 2.90: SGML-Q8a: Implementation in Xcerpt

```
   CONSTRUCT
2  sections_with_flat_titles [
     all section_title {
4      var Section,
       flat_title_content {
6        all var TitleContent
       }
8    }
   ]
10 FROM
   in {
12   resource { "file:sgml.xml" },
     desc var Section → section [[
14     title {{ desc var TitleContent → /.*/ }}
     ]]
16 }
   END
18
   GOAL
20 result [
       all var Section
22   ]
   FROM
24 sections_with_flat_titles [[
     section_title {
26     var Section → section {{}},
       flat_title_content {{ /.*is SGML.*/ }
28   }
   ]]
30 END
```

Xcerpt does not provide means to directly query for text stretching over markup boundaries. Instead, the first rule in Listing 2.90 constructs an element flat_title_content containing the concatenated value of all text nodes (selected in `TitleContent`) under the title of each

section, stripping all markup contained in the title. On this representation the query can be expressed using normal regular expressions.

The XQuery solution is far more concise and uses an implicit type cast: the **contains** function tests whether a string contains another string. In this case the singleton sequence `.` (i.e., `self::node()`) is the first parameter and therefore casted into a string by computing the string value of the first (and only) node in the sequence. As stated above, XQuery's string value is the concatenation of all (directly or indirectly) contained text nodes, as required in this query.

Listing 2.91: SGML-Q8a: Implementation in XQuery

```
1 <result>
    {
3     doc("sgml.xml")//section[.//title[contains(., "is SGML")]]
    }
5 </result>
```

Query SGML-Q8b:
Same as SGML-Q8a, but the string "is SGML" cannot be interrupted by sub-elements, and must appear in a single text node.

Listing 2.92: SGML-Q8b: Implementation in Xcerpt

```
  GOAL
2   result [ all var Section ]
  FROM
4   in {
      resource { "file:sgml.xml" },
6     desc var Section → section {{
        title {{ desc /.*is SGML.*/ }
8     }}
    }
10 END
```

In this variation, the two solutions are very similar again: In both cases text nodes in the title of a section are search for the string. However, the XQuery solution from Chamberlin *et al.* (2005) restricts the occurance of the string to text nodes directly under a title. A section with a title containing an `emph` with the search for string is not matched by the XQuery solution. This is not justified by the formulation of the query and can be resolved by using `.//title//text()` instead of `.//title/text()`, i.e., by considering all descendants of the title.

Listing 2.93: SGML-Q8b: Implementation in XQuery

```
1 <result>
    {
3     doc("sgml.xml")//section[.//title/text()[contains(., "is SGML")]]
    }
5 </result>
```

2.6.4 SGML-Q9,SGML-Q10: Querying Cross-references

Query SGML-Q9:
Locate all the topics referenced by a cross-reference anywhere in the report (all the topicelements whose topicid attribute value is the same as an xrefid attribute value of any xref element).

The two queries use value-based joins (see Section 2.2.3) to locate elements referenced by or referencing other elements in the document.

Listing 2.94: SGML-Q9: Implementation in Xcerpt

```
   CONSTRUCT
2    result [ all var Topic ]
   FROM
4    in {
       resource { "file:sgml.xml" },
6      and {
         desc var Topic → topic [[
8          attributes  {{
             topicid [ var XRefID ]
10         }}
         ]],
12       desc xref [[
           attributes  {{
14           xrefid [ var XRefID ]
           }}
16       ]]
       }
18   }
   END
```

Neither solution of the first query uses any constructs not yet covered. Note again, how Xcerpt uses in logic-programming style multiple occurences of the same variable for joins, whereas XQuery uses an explicit equality constraint.

Listing 2.95: SGML-Q9: Implementation in XQuery

```
   <result>
2    {
       for $id in doc("sgml.xml")//xref/@xrefid
4      return doc("sgml.xml")//topic[@topicid = $id]
     }
6  </result>
```

Query SGML-Q10:
Locate the closest title preceding the cross-reference (xref) element whose xrefid attribute is top4 (the title element that would be touched last before this xref element when touching each element in document order).

Listing 2.96: SGML-Q10: Implementation in Xcerpt

```
1  CONSTRUCT
   results [
3    last ( all var Title )
   ]
5  FROM
   in {
7    resource { "file:sgml.xml" },
     desc /.*/ [[
9      desc title {{
         var Title
11     }},
       desc xref {{
13       attributes  {{
           xrefid { "top4" }
15       }}
       }}
17   ]]
   }
19 END
```

This query combines a value test on an attribute with a relative position query for the closest preceding title. Both solutions select all preceding titles and than return the last of these in document order. Notice, how Xcerpt uses an ordered pattern to select the preceding titles, whereas XQuery uses the before operator <<. Also, since only a single element is to be returned, no iteration using **for** is used in the XQuery case, but variables are bound with **let**.

Listing 2.97: SGML-Q10: Implementation in XQuery

```
  <result>
2 {
    let $x := doc("sgml.xml")//xref[@xrefid = "top4"],
4      $t := doc("sgml.xml")//title[. << $x]
    return $t[last()]
6 }
  </result>
```

2.7 Querying (Untyped) Text Content: Use Case "STRING--String Search"

The use cases in this Section are in many aspects a continuation of Sections 2.2.5 and 2.6.3. They detail more the kind of content queries expected from XML query languages.

The queries are based on two data sets with large text fragments. Both are fairly regular and flat (maximum depth 6) with non-recursive structure definitions. Almost all element types can only occur under elements of a single type.

Listing 2.98: Excerpt of a document containing news items about companies (referrenced as `string.xml`)

```
1 <?xml version="1.0" encoding="ISO−8859−1"?>
  <news>
3 <news_item>
    <title> Gorilla Corporation acquires YouNameItWeIntegrateIt.com </title>
5   <content>
      <par> Today, Gorilla Corporation announced that it will purchase
7        YouNameItWeIntegrateIt.com. The shares of
         YouNameItWeIntegrateIt.com dropped $3.00 as a result of this
9        announcement.
      </par>
11
      <par> As a result of this acquisition, the CEO of
13       YouNameItWeIntegrateIt.com Bill Smarts resigned. He did not
         announce what he will do next.  Sources close to
15       YouNameItWeIntegrateIt.com hint that Bill Smarts might be
         taking a position in Foobar Corporation.
17    </par>

19    <par>YouNameItWeIntegrateIt.com is a leading systems integrator
         that enables <quote>brick and mortar</quote> companies to
21       have a presence on the web.
      </par>
23
    </content>
25  <date>1-20-2000</date>
    <author>Mark Davis</author>
27  <news_agent>News Online</news_agent>
  </news_item>
29
  [...]
31 </news_item>
  </news>
```

Listing 2.99: Excerpt of a document on companies and their partners and competitors (referenced as `company-data.xml`)

```
1  <company>
     <name>Foobar Corporation</name>
3    <ticker_symbol>FOO</ticker_symbol>

5    <description>Foobar Corporation is a maker of Foo(TM) and
           Foobar(TM) products and a leading software company with a 300
7          Billion dollar revenue in 1999. It is located in Alaska.
     </description>

9
     <business_code>Software</business_code>
11   <partners>
           <partner>YouNameItWeIntegrateIt.com</partner>
13         <partner>TheAppCompany Inc.</partner>
     </partners>
15   <competitors>
           <competitor>Gorilla Corporation</competitor>
17   </competitors>
   </company>
```

STRING-Q1 and STRING-Q3 are omitted, for Q1 is straightforward and does not exhibit any constructs not discussed in previous sections and Q3 has been dropped from Chamberlin *et al.* (2005).

2.7.1 STRING-Q2,STRING-Q4: Value-based Approximate Joins

The queries in this section are focused on two issues:

· Boolsche keyword queries: An element is selected based on a boolean expression on occurrence of keywords in its content. Such queries are typical for information retrieval systems and, though supported in their basic form by Xcerpt and XQuery, not easily expressed in either of them. E.g., issues such as ranking, approximation, stemming, text regions, etc. are not covered by either language at the moment (and not required for the basic queries in this section).

· Finding the value of one element in the content of another (a form of *approximate join*). This is a form of content query that in fact occurs often in an XML context (e.g., when handling ID/IDREF links where the value of an ID attribute is referenced in an IDREF attribute which might contain additional references to other ID attributes).

Query STRING-Q2:
Find news items where the Foobar Corporation and one or more of its partners are mentioned in the same paragraph and/or title. List each news item by its title and date.

Listing 2.100: STRING-Q2: Implementation in Xcerpt

```
1  GOAL
   result [
3    all news_item [
       var Title,
5      var Date
     ]
7  ]
   FROM
9  and {
     in {
```

```
11      resource { "file:company_data.xml" },
        company {{
13        name {"Foobar Corporation"},
          desc partner {var Partner}
15      }}
      },
17    in {
        resource { "file:string.xml" },
19      desc news_item {{
          var Date → date {{ }},
21        and {
            var Title → title {{ }},
23          desc /title|par/ {{
              and { desc /.*Foobar Corporation.*/,
25                  desc /.*(var Partner → .*).*/ }
            }}
27        }
        }}
29    }
    }
31  END
```

The Xcerpt solution uses, as in the previous containment queries, regular expressions to find the sought for keywords in the content of an element. Regular expressions are also used to specify the label of the elements in whose content to search (/title|par/). The co-occurrence of the keywords is expressed by an **and** around the two regular expressions. Without the outer **and** an element matched by lines 23-26 would be required to be different from an element matched by line 22. A limitation of the Xcerpt solution is that the keywords may not be interspersed by markup.

The XQuery solution uses, for readability and re-usability, a separate, parameterized function to computer the partners of a company. In contrast to proper value-based joins, where the existentially quantified semantics of XQuery's = (called a "general" comparison) comes in handy, for approximate joins the existential quantification must be explicitly stated using **some … satisfies**. Of course, also proper value-based joins can be expressed this way, as $a = b$ is (modulo type coercion) equivalent to some $el_of_a in a satisfies some $el_of_b in b satisfies $el_of_a eq $el_of_b. The explicit existential quantification makes the XQuery solution very verbose.

Listing 2.101: STRING-Q2: Implementation in XQuery

```
    declare function local:partners($company as xs:string) as element()*
2   {
        let $c := doc("company–data.xml")//company[name = $company]
4       return $c//partner
    };
6
    let $foobar_partners := local:partners("Foobar Corporation")
8
    for $item in doc("string.xml")//news_item
10  where
      some $t in $item//title satisfies
12      (contains($t/text(), "Foobar Corporation")
        and (some $partner in $foobar_partners satisfies
14        contains($t/text(), $partner/text())))
      or (some $par in $item//par satisfies
16     (contains(string($par), "Foobar Corporation")
         and (some $partner in $foobar_partners satisfies
18         contains(string($par), $partner/text()))))
    return
20      <news_item>
            { $item/title }
22          { $item/date }
        </news_item>
```

Query STRING-Q4:
Find news items where a company and one of its partners is mentioned in the same news item and the news item is not authored by the company itself.

Listing 2.102: STRING-Q4: Implementation in Xcerpt

```
   CONSTRUCT
 2 result {
     all var News_Item
 4 }
   FROM
 6 and {
     in {
 8     resource { "file:company_data.xml" },
       company {{
10       name { var Company },
         desc partner { var Partner }
12     }}
     },
14   in {
       resource { "file:string.xml" },
16     desc var News_Item → news_item {{
         and {
18         desc /.*(var Company → .*).*/,
           desc /.*(var Partner → .*).*/
20       },
         without news_agent { var Company }
22     }}
     }
24 }
   END
```

Xcerpt's solution is very similar to the previous one, the main differences being the **without** construct indicating that no `news_agent` element with the company's name as value may occur in the news item.

XQuery's solution differs here as it only tests that one of the `news_agent`'s values are different (as the **!=** has existentially quantified semantics). If multiple `news_agent` elements might occur, a **every** expression must be used, further complicating the query.

Listing 2.103: STRING-Q4: Implementation in XQuery

```
   declare function local:partners($company as xs:string) as element()*
 2 {
       let $c := doc("company–data.xml")//company[name = $company]
 4     return $c//partner
   };
 6
   for $item in doc("string.xml")//news_item,
 8     $c in doc("company–data.xml")//company
   let $partners := local:partners($c/name)
10 where contains(string($item), $c/name)
     and (some $p in $partners satisfies
12     contains(string($item), $p) and $item/news_agent != $c/name)
   return
14     $item
```

2.7.2 STRING-Q5: String Concatenation

Query STRING-Q5:
For each news item that is relevant to the Gorilla Corporation, create an "item summary" element. The content of the item summary is the content of the title,

date, and first paragraph of the news item, separated by periods. A news item is relevant if the name of the company is mentioned anywhere within the content of the news item.

The final query in this part is concerned about concatenating the value of multiple elements.

Listing 2.104: STRING-Q5: Implementation in Xcerpt

```
1  CONSTRUCT
   result {
3    all item_summary [
       var Title,
5      ".",
       var Date,
7      ".",
       first ( all var Par )
9    ]
   }
11 FROM
   in {
13   resource { "file:string.xml" },
     desc news_item {{
15     content {{
         and [
17         desc var G → /.*Gorilla Corporation.*/,
           desc var Par → par {{}}
19       ]
       }},
21     title { var Title },
       date { var Date }
23   }}
   }
25 END
```

In Xcerpt, basic string concatenation can be obtained simply by having several text nodes consecutive children of the same parent. Such text nodes are collapsed at construction into a single one by concatenation. This provides a very natural mean for string concatenation. More complex constructions of strings might require additional string processing functions that are not yet finalized.

The XQuery solution uses the function **concat** to concatenate the values of title and date with a following dot (which implicitly computes their string value, thus requiring the explicit computation of the string value only for the first paragraph).

Listing 2.105: STRING-Q5: Implementation in XQuery

```
  for $item in doc("string.xml")//news_item
2 where contains(string($item/content), "Gorilla Corporation")
  return
4     <item_summary>
          { concat($item/title,". ") }
6         { concat($item/date,". ") }
          { string(($item//par)[1]) }
8     </item_summary>
```

2.8 Querying Namespaces: Use Case "NS--Queries using Namespaces"

An important aspect of XML documents has been ignored by the previous use cases: namespaces. In this section, another data set on auctions is used that makes heavy use of names-

71

paces, e.g., to indicate rating and user information from different auction sites or to specify links using W3C's linking language XLink (DeRose *et al.* 2001).

The namespace information in that data is queried in different ways by the queries in this section: some select the namespace URIs in the document, some use the namespaces to find the intended information.

Listing 2.106: Excerpt of the auction data (referred to as `auction.xml`)

```
1  <ma:AuctionWatchList
       xmlns:ma="http://www.example.com/AuctionWatch"
3      xmlns:xlink="http://www.w3.org/1999/xlink"
       xmlns:anyzone="http://www.example.com/auctioneers#anyzone"
5      xmlns:eachbay="http://www.example.com/auctioneers#eachbay"
       xmlns:yabadoo="http://www.example.com/auctioneers#yabadoo" >
7  <ma:Auction anyzone:ID="0321K372910">
       <ma:AuctionHomepage
9          xlink:type="simple"
           xlink:href="http://www.example.com/item/0321K372910" />
11     <ma:Schedule>
           <ma:Open    xmlns:dt="http://www.w3.org/2001/XMLSchema"
13              dt:type="timeInstant">2000-03-21:07:41:34-05:00</ma:Open>
           <ma:Close   xmlns:dt="http://www.w3.org/2001/XMLSchema"
15              dt:type="timeInstant">2000-03-23:07:41:34-05:00</ma:Close>
       </ma:Schedule>
17     <ma:Price>
           <ma:Start ma:currency="USD">3.00</ma:Start>
19         <ma:Current ma:currency="USD">10.00</ma:Current>
           <ma:Number_of_Bids>5</ma:Number_of_Bids>
21     </ma:Price>
       <ma:Trading_Partners>
23         <ma:High_Bidder>
               <eachbay:ID>RecordsRUs</eachbay:ID>
25             <eachbay:PositiveComments>231</eachbay:PositiveComments>
               <eachbay:NeutralComments>2</eachbay:NeutralComments>
27             <eachbay:NegativeComments>5</eachbay:NegativeComments>
               <ma:MemberInfoPage
29                 xlink:type="simple"
                   xlink:href="http://auction.eachbay.com/members?get=RecordsRUs"
31                 xlink:role="ma:MemberInfoPage"
               />
33         </ma:High_Bidder>
           <ma:Seller>
35             <anyzone:ID>VintageRecordFreak</anyzone:ID>
               <anyzone:Member_Since>October 1999</anyzone:Member_Since>
37             <anyzone:Rating>5</anyzone:Rating>
               <ma:MemberInfoPage
39                 xlink:type="simple"
                   xlink:href="http://auction.anyzone.com/members/VintageRecordFreak"
41                 xlink:role="ma:MemberInfoPage"
               />
43         </ma:Seller>
       </ma:Trading_Partners>
45
       <ma:Details>
47         <record xmlns="http://www.example.org/music/records">
               <artist>Miles Davis</artist>
49             <title>In a Silent Way</title>
               <recorded>1969</recorded>
51             <label>Columbia Records</label>

53             <remark>
                   With Miles Davis (trumpet), Herbie Hancock (Electric
55                 Piano), Chick Corea (Electric Piano), Wayne Shorter
                   [...]
57             </remark>

59         </record>
       </ma:Details>
```

```
61
   </ma:Auction>
63 [...]
   </ma:Auction>
65 </ma:AuctionWatchList>
```

2.8.1   NS-Q1: Selection of Namespace URIs

Query NS-Q1:
List all unique namespaces used in the sample data.

The first query demonstrates how to select the value (i.e., the URI) of namespace nodes in a query.

In Xcerpt all elements are labeled with a 2-tuple of namespace and tag name. E.g., `ma:Seller {{ }}` matches elements with tag name `Seller` and the namespace bound to the namespace prefix `ma`. Like tag names namespaces can be queried using regular expressions. If *no* namespace is given, elements with the appropriate tag name in all namespaces match, e.g., `Seller {{ }}` is an abbreviation for `/.*/:Seller {{ }}`. *Namespace prefixes* are merely abbreviations. Therefore, the current handling of namespaces in Xcerpt does not distinguish between two elements with same tag name and namespace but different namespace prefix. Also, *namespace URIs* can be used directly instead of defining a namespace prefix, e.g., `"http://www.example.com/AuctionWatch":Seller {{ }}`.

Listing 2.107: NS-Q1: Implementation in Xcerpt

```
   GOAL
 2 Q1 {
     all ns {
 4     var NSUri
     }
 6 }
   FROM
 8 in {
     resource { "file:auction.xml" },
10   desc var NSUri:/.*/ {{}}
   }
12 END
```

Listing 2.107 shows how to collect all namespace URIs occurring in a document. As stated before, namespaces are always treated as URIs, with prefixes as insignificant abbreviations. Therefore it is, e.g., not possible to select all namespace prefixes in a document.

XQuery's namespace handling differs quite notably: First unqualified elements are considered to be in the default element namespace, e.g., `child::a` selects only those `a` children in the default element namespace. If not set otherwise the default element namespace is the empty namespace. To select all `a` children regardless of their namespace the expression `child::*:a` must be used. Second, as tag names, namespace URIs and prefixes must be collected using special functions (**namespace-uri** and **prefix-from-QName**). Third, as indicated by the **prefix-from-QName** function, XQuery preserves namespace prefixes. This means in particular that the string representation of names (as returned by the `name` function) of two nodes with the same tag name (called "local" name in XQuery) and namespace URI may differ, if they use different namespace prefixes for the same namespace URI. However, it allows to return all namespace prefixes used in a document, e.g., in Listing 2.108 by changing line 5 to `return prefix-from-QName(name($i))`.

73

```
1 <Q1>
    {
3     for $n in distinct-values(
                for $i in (doc("auction.xml")//* | doc("auction.xml")//@*)
5               return namespace-uri($i)
            )
7     return  <ns>{$n}</ns>
    }
9 </Q1>
```

2.8.2 NS-Q2 to NS-Q4: Selection based on Namespaces

The three queries of this section are concerned with selection where some of the selected elements or attributes have namespaces.

Query NS-Q2:
Select the title of each record that is for sale.

As visible in the sample data from Listing 2.106, the title element is in the namespace `http://www.example.org/music/records`. Thus both solutions define a namespace prefix (`music`) and use that prefix to select only elements with tag title in the `music` namespace.

Listing 2.109: NS-Q2: Implementation in Xcerpt

```
  ns-prefix music = "http://www.example.org/music/records"
2
  GOAL
4 Q2 {
    all var Title
6 }
  FROM
8 in {
    resource { "file:auction.xml" },
10  desc var Title → music:title {{}}
  }
12 END
```

Listing 2.110: NS-Q2: Implementation in XQuery

```
1 declare namespace music = "http://www.example.org/music/records";

3 <Q2>
    {
5     doc("auction.xml")//music:title
    }
7 </Q2>
```

Query NS-Q3:
Select all elements that have an attribute whose name is in the XML Schema namespace.

A variation of query NS-Q3, the query requires to query for attributes in a certain namespace regardless of their tag. Instead of special wildcard operators as XQuery, Xcerpt uses, as detailed in previous sections, regular expressions for such queries.

Listing 2.111: NS-Q3: Implementation in Xcerpt

```
   ns-prefix dt="http://www.w3.org/2001/XMLSchema"
2
   GOAL
4  Q3 {
     all var Element
6  }
   FROM
8  in {
     resource { "file:auction.xml" },
10   desc var Element → /.*/ {{
       attributes {{
12       dt:/.*/ {{}}
       }}
14   }}
   }
16 END
```

Listing 2.112: NS-Q3: Implementation in XQuery

```
1  declare namespace dt = "http://www.w3.org/2001/XMLSchema";

3  <Q3>
     {
5      doc("auction.xml")//*[@dt:*]
     }
7  </Q3>
```

Query NS-Q4:
List the target URI's of all XLinks in the document.

More precisely, the query's result should be a list of ns elements with href attributes in the XLink namespace. Both solutions find all elements in the document and then select only those attributes matching the condition. Recall, that XQuery's `//` is an abbreviation for `/descendant-or-self::node()/`.

Listing 2.113: NS-Q4: Implementation in Xcerpt

```
   ns-prefix xlink="http://www.w3.org/1999/xlink"
2
   GOAL
4  Q4 {
     all ns {
6      var Href
     }
8  }
   FROM
10 in {
     resource { "file:auction.xml" },
12   desc /.*/ {{
       attributes {{
14       var Href → xlink:href {{}}
       }}
16   }}
   }
18 END
```

Listing 2.114: NS-Q4: Implementation in XQuery

```
1  declare namespace xlink = "http://www.w3.org/1999/xlink";

3  <Q4 xmlns:xlink="http://www.w3.org/1999/xlink">
     {
```

```
5    for $hr in doc("auction.xml")//@xlink:href
       return <ns>{ $hr }</ns>
7    }
   </Q4>
```

2.8.3 NS-Q8: Ignoring Namespaces

Query NS-Q8:
Select all traders (either seller or high bidder) without negative comments.

The final query on namespaces covers the case, where the tag name of an element is known, but not its namespace.

Listing 2.115: NS-Q8: Implementation in Xcerpt

```
1  ns-prefix ma = "http://www.example.com/AuctionWatch"

3  GOAL
   Q4 {
5    all ns {
       var Trader
7    }
   }
9  FROM
   in {
11   resource { "file:auction.xml" },
     desc ma:Trading_Partners {{
13     var Trader → ma:/Seller|High_Bidder/ {{
         without /.*/:NegativeComments { var NrNegComments } where NrNegComments != 0
15   }}
   }
17 END
```

The Xcerpt solution differs from the XQuery solution in the handling of multiple NegativeComments elements. Although not occurring in the data, this it is possible for a single trader to have comments from different auction sites, thus multiple NegativeComments children. In the XQuery case a single 0 valued such child suffices to select the trader, whereas the Xcerpt solution that there is no NegativeComments element with value different from 0. The latter behavior can be obtained in XQuery by replacing line 8 with where not($s/*:NegativeComments != 0) (double negation) or by using a corresponding **every … satisfies** construct.

Notice, how the XQuery solution declares a number of namespace prefixes explicitly on the Q8 element (as also in NS-Q4). Both Xcerpt and XQuery ensure proper namespace serialization (e.g., by defining prefixes where necessary). However, explicitly stated prefixes are taken into consideration during serialization thus allowing the user to select place and prefix name.

Listing 2.116: NS-Q8: Implementation in XQuery

```
   declare namespace ma = "http://www.example.com/AuctionWatch";
2
   <Q8 xmlns:ma="http://www.example.com/AuctionWatch"
4      xmlns:eachbay="http://www.example.com/auctioneers#eachbay"
       xmlns:xlink="http://www.w3.org/1999/xlink">
6    {
       for $s in doc("auction.xml")//ma:Trading_Partners/(ma:Seller | ma:High_Bidder)
8      where $s/*:NegativeComments = 0
       return $s
10   }
   </Q8>
```

2.9 Restructuring: Use Case "PARTS--Recursive Parts Explosion"

The final use case from Chamberlin *et al.* (2005) covered here is the dual to the case of Section 2.3.2: It constructs a tree structure given a flat list of records with non-resolved references among the records. From the flat data on parts of a car shown in Listing 2.117 the hierarchical representation in Listing 2.118 should be produced.

Listing 2.117: Flat part data (referred to as `partlist.xml`)

```
<?xml version="1.0" encoding="ISO−8859−1"?>
2 <partlist>
   <part partid="0" name="car"/>
4  <part partid="1" partof="0" name="engine"/>
   <part partid="2" partof="0" name="door"/>
6  <part partid="3" partof="1" name="piston"/>
   <part partid="4" partof="2" name="window"/>
8  <part partid="5" partof="2" name="lock"/>
   <part partid="10" name="skateboard"/>
10 <part partid="11" partof="10" name="board"/>
   <part partid="12" partof="10" name="wheel"/>
12 <part partid="20" name="canoe"/>
</partlist>
```

Listing 2.118: Resulting hierarchical part data

```
1 <parttree>
    <part partid="0" name="car">
3       <part partid="1" name="engine">
            <part partid="3" name="piston"/>
5       </part>
        <part partid="2" name="door">
7           <part partid="4" name="window"/>
            <part partid="5" name="lock"/>
9       </part>
    </part>
11  <part partid="10" name="skateboard">
        <part partid="11" name="board"/>
13      <part partid="12" name="wheel"/>
    </part>
15  <part partid="20" name="canoe"/>
</parttree>
```

Query PARTS-Q1:
Convert the sample document from partlist *format to* parttree *format (see Listing 2.118). In the result document, part containment is represented by containment of one* part *element inside another. Each part that is not part of any other part should appear as a separate top-level element in the output document.*

In Xcerpt this query can be expressed in two different ways: First by using Xcerpt's graph references and second using a recursive transformation function as in the XQuery solution. The first approach is shown in Listing 2.119, the second in Listing 2.120.

Listing 2.119: PARTS-Q1: Implementation in Xcerpt using graph references

```
  GOAL
2 parttree [
    all var TopLevelPart
4 ]
  FROM
6 all-top-parttree [[
    var TopLevelPart → part {{ }},
```

```
 8    without part {{
        desc var TopLevelPart → part {{ }}}
10    }}
    ]]
12 END

14 CONSTRUCT
   all-top-parttree [
16    all var PartId@part {
        attributes {
18        partid { var PartId },
          name { var Name }
20      },
        optional all ^var SubPartId
22    }
   ]
24 FROM
   in {
26    resource { "file:partlist.xml", "xml" },
      partlist {{
28      part {
          attributes {{
30          partid { var PartId },
            name { var Name }
32        }}
        },
34      optional part {
          attributes {{
36          partid { var SubPartId },
            partof { var PartId },
38        }}
        }
40    }}
   }
42 END
```

Using graph references a part element in the parttree is constructed by copying the two attributes and creating a reference to each of the part's sub-parts, if there are any. Notice the use of **optional** to indicate that there may or may not be any sub-parts. Graph references are expressed using @ to define the end point of a reference and ^ to link to a defined end point. As discussed above, links created using this mechanism can be transparently queried using Xcerpt's term matching. In the case of this query some post-processing of the created structure is necessary, as all parts are also direct children of the root all-top-parttree, but the query asks that only parts that are not sub-part of another part are such children. Therefore the first rule collects all such parts and constructs a new tree with the correct structure.

The second solution in Xcerpt is similar to the solution in XQuery: it uses a recursive transformation rule (in XQuery: function) to map the flat representation of each part to its hierarchical representation (i.e., with sub-parts nested inside). For more details on these solutions consult Section 2.3.1.

Listing 2.120: PARTS-Q1: Implementation in Xcerpt using a recursive transformation rule

```
1 GOAL
  parttree [
3    all var TransformedTopLevelPart
  ]
5 FROM
  and {
7    in {
      resource { "file:partlist.xml", "xml" },
9      partlist [[
        var TopLevelPart → part {
11        without attributes {{ partof {{}} }}
        }
```

```
13     ]]
     },
15   transform [ var TopLevelPart, var TransformedTopLevelPart }
   }
17 END

19 CONSTRUCT
   transform [
21   var PartInList,
     part [
23     attributes {
         partid { var PartID },
25       name { var Name }
       },
27     optional all var TransformedSubPart
     ]
29 ]
   FROM
31 and [
     in {
33     resource { "file:partlist.xml", "xml" },
       partlist {{
35       var PartInList → part {
           attributes {{
37           partid { var PartId },
             name { var Name }
39         }}
         },
41       optional var SubPart → part {
           attributes {{
43           partof { var PartId }
           }}
45       }
       }}
47   },
     transform [ var SubPart, var TransformedSubPart ]
49 ]
   END
```

Listing 2.121: PARTS-Q1: Implementation in XQuery

```
1 declare function local:one_level($p as element()) as element()
   {
3     <part partid="{ $p/@partid }"
           name="{ $p/@name }" >
5         {
               for $s in doc("partlist.xml")//part
7             where $s/@partof = $p/@partid
             return local:one_level($s)
9         }
     </part>
11 };

13 <parttree>
     {
15   for $p in doc("partlist.xml")//part[empty(@partof)]
     return local:one_level($p)
17 }
   </parttree>
```

Chapter 3

Querying RDF: Realizing the W3C RDF Data Access Use Cases in Xcerpt

3.1 Introduction and Preliminaries

Where the previous part has considered the W3C's XML Query Use Cases, this part shows Xcerpt's ability to query RDF on a select few of the use cases presented in the W3C's Data Access Use Cases Clark (2004). Not the full set of use cases is considered at the moment as Clark (2004) is still very much work in progress. E.g., most use cases are currently only specified in rather vague natural language without sample data or expected result. Also a number of use cases seems to vary only slightly or with respect to secondary issues not relevant for the consideration of query languages.

The reader should also be aware that the access to RDF in Xcerpt is still under active development and might change in several aspects, e.g., with regard to the handling of URI's and typed literals. Also additional constructs for easing RDF access are under consideration, see Bolzer (2005).

3.1.1 Querying RDF in Xcerpt: A Matter of View(point)

This section gives a brief introduction into querying RDF with Xcerpt starting with a short overview of RDF and RDF Schema, the dominant formats for meta-data in the (Semantic) Web.

RDF and RDF Schema: Metadata Representation in the Semantic Web

RDF Klyne *et al.* (2004) data is sets of "triples" or "statements" of the form (*Subject*, *Property*, *Object*)$. RDF data is commonly seen as a directed graph, whose nodes correspond to a statement's subject and object and whose arcs correspond to a statement's property (thus relating a subject with an object). Nodes (i.e., subjects and objects) are labeled by either (1) URIs describing (Web) resources, or (2) literals (i.e., scalar data such as strings or numbers), or (3) are unlabeled, being so-called anonymous or "blank nodes". Blank nodes are commonly used to group or "aggregate" properties. Edges (i.e., Properties) are always labeled by URIs indicating the type of relation between its subject and object.
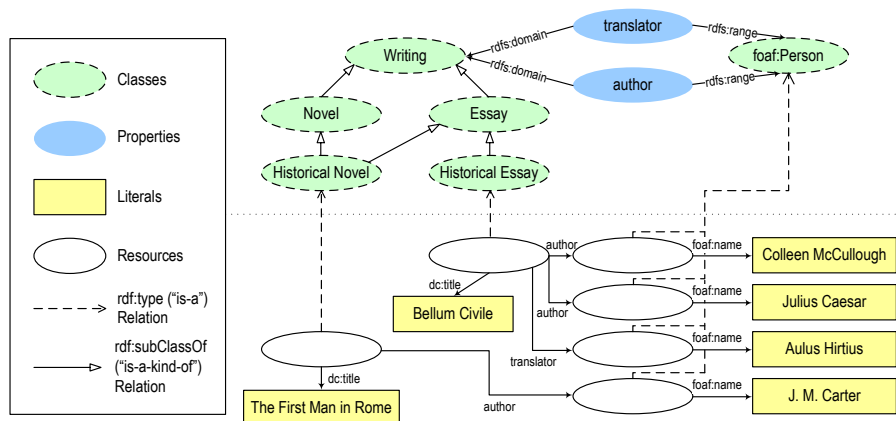
Figure 3.1: Sample Data: representation as a (simplified) RDF graph

RDFS allows one to define so-called "RDF Schemas" or "ontologies", similar to object-oriented data models. Based on an RDFS, inference rules can be specified, for instance the transitivity of the class hierarchy, or the type of an untyped resource that has a property associated with a known domain.

RDF can be *serialized* in various formats, the most frequent being XML. Early approaches to RDF serialisation have raised considerable criticism due to their complexity. As a consequence, a surprisingly large number of RDF serialisation have been proposed, cf. Furche *et al.* (2004).

Figure 3.1 shows the running example for this article, a (simplified) representation of an RDF graph as used, e.g., in a book recommender system.

The remainder of this section presents two different perspectives on RDF: (1) a flat, almost relational view and (2) a graph view reminiscent of semi-structured data. These perspectives are compared briefly with some existing approaches for RDF querying.

To illustrate these two perspectives, the selection query "Select all Essays together with their authors (i.e., author URIs and corresponding names)" is used against the data of Figure 3.1.

RDF Triples: A Flat, Relational View

The following Xcerpt program expresses the above query on a triple view of the RDF data:

```
   DECLARE ns-prefix rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 2 DECLARE ns-prefix books = "http://example.org/books#"
   GOAL
 4   result [
       all essay [
 6       id [ var Essay ],
         all author [
 8         id [ var Author ],
           all name [ var AuthorName ]
10   ] ] ]
   FROM
12   and{ RDFS-TRIPLE [
         var Essay, rdf:type{{}}, books:Essay{{}}          ],
14     RDF-TRIPLE [
         var Essay, books:author{{}}, var Author            ],
```

```
16    RDF-TRIPLE [
         var Author, books:authorName{{}}, var AuthorName   ]  }
18 END
```

The query pattern (between **FROM** and **END**) is a conjunction of queries against the RDF triples represented in the predicate RDF-TRIPLE using the prefixes declared in line 1 and 2. Notice that the first conjunct actually uses RDFS-TRIPLE. This view of the RDF data contains all basic triples plus the ones entailed by the RDFS semantics (cf. Bolzer (2005) for a detailed description). Using `RDFS-TRIPLE` instead of `RDF-TRIPLE` ensures that also resources actually classified in a sub-class of `books:Essay` are returned.

In the construct pattern (between **GOAL** and **FROM**), one of the strengths of combining XML and RDF querying in Xcerpt is shown: Following the W3C's requirements for an RDF data access language, yet in contrast to most other RDF query languages, it is possible to construct arbitrary XML: E.g., here, a list of all essays with their authors grouped inside is constructed.

Except for the construction of arbitrary XML, a similar (triple) view of RDF is taken in most of the current RDF query languages, most notably in RDQL and the W3C's SPARQL Prud'hommeaux and Seaborne (2005), and also in Robie (2001), an approach for querying RDF with XQuery: A query is composed of conjunctions (and in some languages including our proposal disjunctions) of "triple patterns", i.e., triples with variables indicating queried data. Using multiple occurrences of same variables more complex conditions can be expressed, e.g., for traversing paths in the RDF data or even for restricting a resource using several of its properties. While familiar from SQL, this style leads for RDF data to hard-to-read and lengthy queries that also pose problems for evaluation (cf., e.g., Hung *et al.* (2005)).

The previous observations lead us to an alternative view of RDF that is both closer to its actual data model and can make better use of the advanced features of an XML query language such as the traversal of arbitrary length paths in tree or graph data.

RDF Graph: A Semi-structured View

For this view of RDF, Xcerpt's treatment of XML as graph data is an advantage over XML query languages such as XPath or XQuery, which consider XML as strictly tree shaped, providing no direct support for (ID/IDREF or similar) links in the data model. Although there have been proposals for slicing an (acyclic) RDF graph into trees for processing them with XSLT or XQuery (e.g., Walsh (2003)), these approaches invariantly suffer (a) from choosing an appropriate slicing and (b) from the (in general) exponential blow-up of the tree view of an acyclic RDF graph.

In Xcerpt, a graph view of RDF is rather natural as the following Xcerpt program expressing the same query as above, but on the graph instead of the triple view, demonstrates:

```
1   ... % prefixes and construction identical to above query
  FROM
3   RDFS-GRAPH {{
       var Essay {{
5        rdf:type {{ books:Essay {{ }} }},
         books:author {{
7          var Author {{ books:name {{ var AuthorName }} }}
         }}
9     }} }}
  END
```

The RDF graph view is represented in the RDF-GRAPH predicate. Here, the RDFS-GRAPH view is used that extends RDF-GRAPH as RDFS-TRIPLE extends RDF-TRIPLE. Triples are represented

similar to striped RDF/XML: each resource is a direct child element in RDF-GRAPH with a sub-element for each statement with that resource as object. The sub-element is labeled with the URI of the predicate and contains the object of the statement. As Xcerpt's data model is a rooted *graph* this can be represented without duplication of resources.

In contrast to the previous query against the RDF triple view, no conjunction is used but rather a nested pattern that naturally reflects the structure of the RDF graph. The more complex a query gets, the more evident the advantage of the graph view becomes: instead of having to use multiple occurrences of same variables for relating parts of the query, that relation is represented in the structure of the query itself (represented in the textual version of the query shown above by nesting and indentation).

Path traversals of arbitrary length can be expressed using traversal operators such as descendant. E.g., to find all subclasses of a given class one can use Xcerpt's qualified descendant `desc(rdfs:subClassOf<rdfs:Class)*` that is similar to regular path expressions or conditional XPath. Similarly, other constructs for querying XML data with incomplete information about the structure of the queried data can be used for RDF as well.

Considering the efficient evaluation of queries against such a graph view of RDF data, there are results on the efficient evaluation of queries against graph-shaped semi-structured data, cf. Schenkel *et al.* (2004). Ongoing work by the authors targets efficient evaluation methods for implementing Xcerpt queries against graph-shaped data. We believe it likely that at least for some interesting subsets of Xcerpt queries efficient evaluation methods against graph-shaped data can be found.

A "Retrospective" View: From Triples To Graphs

A final observation on the graph view is that it can not only be implemented directly on the different RDF serializations (just as the triple view) but also on top of the triple view, as shown in the following rule:

```
1  CONSTRUCT
     RDF-GRAPH {
3      all var Subject @ var Subject:var SubjectType {
         all optional var Predicate {  ^var Object   },
5        all optional var Predicate {  var Literal   }
       } }
7  FROM
     or{
9      RDF-TRIPLE[
         var Subject,  var Predicate:uri{},
11       optional var Literal as literal{{}},
         optional var Object {{}} where { var Object != 'literal'}
13     ],
       RDF-TRIPLE[
15       /.*/:/.*/{{}},  /.*/:/.*/{{}},  var Subject{{}}
       ]  }
17 END
```

Notice the use of the **optional** keyword in lines 11 and 12. This indicates that the contained part of the pattern does not have to occur in the data, but if it does occur the contained variables are bound appropriately. In lines 3 and 4 the actual graph structure is constructed: by using the operators @ and ^ a (possibly cyclic) link can be constructed.

3.1.2 The RDF Data Access Use Cases Specification

The RDF Data Access Work Group of the W3C has collected a large number of such use cases for the evaluation of the RDF query language SPARQL it is developing. As these use cases overlap in many parts, only a small selection is described here. The use cases are roughly reordered based on their complexity, guardedly introducing more and more aspects, such as querying of multiple sources and inferencing.

The sample data used for many of the use cases are verbatim copies of the samples of the DAWG Use Cases, others have been generated specifically for this section. Following the DAWG, the data is presented in Notation 3.

3.2 Basic Selection: Finding an Email Address (Personal Information Management)

Despite its original positioning as a meta data format, RDF is increasingly used for immediate representation of data. The use case envisions storage of an address book in RDF, using the FOAF (see \url{http://www.foaf-project.org/}) vocabulary for description of personal information and relationships. The whole address book is stored as a single RDF graph, with each entry of the address book represented by a blank node and a number of statements directly describing the node.

Listing 3.1: Sample E-Mail Adress represented in RDF

```
  @prefix foaf:  &lt;http://xmlns.com/foaf/0.1/&gt; .
2 []
      foaf:name    "Johnny Lee Outlaw" ;
4     foaf:mbox    &lt;mailto:jlow@example.com&gt; .
```

The address book is queried by specifying values for one or more properties, returning the value of another property of entries matching the query. In the given use case, the name of a person is specified in order to query his/her e-mail address, returning a list of matches.

Query RDF-Q1:

```
1 GOAL
    email{ all address{ var MAILTO } }
3 RDF{{
    /.*/:/.*/{{
5     "http://xmlns.com/foaf/0.1/name":uri{ literal{"Johnny Lee Outlaw"} },
      "http://xmlns.com/foaf/0.1/mbox":uri{ var MAILTO:uri{{}} }
7   }}
  }}
9 END
```

This query is formulated against the graph representation of RDF graphs. It specifies a node of arbitrary type and identifier with two properties, a name and an e-mail address. The name is specified as a literal value, while the e-mail address, which is an URI, is unknown and bound to the variable MAILTO. The construction term of the rule has been omitted, as the use cases does not specify a specific result form.

```
  GOAL
2   email{ all address{ var MAILTO } }
```

```
   FROM
 4   and{
       RDF-TRIPLE[ var X, "http://xmlns.com/foaf/0.1/name":uri{}, literal{"Johnny Lee Outlaw"} ],
 6     RDF-TRIPLE[ var X, "http://xmlns.com/foaf/0.1/mbox":uri{}, var MAILTO:uri{} ]
     }
 8 END
```

The second query queries not the graph representation but triple representation of the same graph. In it, two triple patterns are joined using the common variable *X*.

3.3   Basic Combination: Finding Information about Motorcycle Parts (Supply Chain Management)

The use case describes data retrieval from a parts database that includes dependency information between individual parts, stored as a single RDF graph.

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

 5 <http://triumph.example/part/0d92ie433>
       rdf:type       triumph:part ;
 7     rdfs:label     "Accelerator Cable MK3" ;
       triumph:depends-on  <http://triumph.example/part/329i2dk39> ;
 9     triumph:part-for    <http://triumph.example/2004/SpeedTriple> ;
       triumph:part-number "LCD 100-04BSPT" .
11
   <http://triumph.example/part/329i2dk39>
13     rdfs:label     "Mounting Bracket" ;
       triumph:requires
15         [ triumph:has-number  "4" ;
             triumph:part-number  "149028ab-MT" ;
17           triumph:type   triumph:screwx
           ] .
```

A query is sought that retrieves the human-readable description of a part, given it's URI, and all dependent parts that must be replaced simultaneously. The use case differs from the previous one in that it requests information about more than one resource. A simple HTML page that is displayed to the user is constructed as the result of the query. The function **join** is used to concatenates multiple string literals into a single literal subterm.

```
 1  html{
       head{ title{ "Search Results" } },
 3     body{
         h1{ join( var PART_LBL, "(", var PART_NUM, ")" },
 5       p{ "Part for:" },
         ul{
 7         all li{ var VEHICLE_LBL }
         },
 9       h2{ "Dependencies:" },
         optional ul{
11         all li{
             join( var DEP_LBL, "(", var DEP_NUM, ")" )
13         }
         } with default "-"
15     }
     }
17 FROM
     RDF{{
19     "http://triumph.example/part/0d92ie433":uri{{
         "http://www.w3.org/2000/01/rdf-schema#label":uri{
21         literal{ var PART_LBL }
```

```
      },
23    "http://triumph.example/schema/#part-number":uri{
        literal{ var PART_NUM }
25    },
      "http://triumph.example/schema/#part-for":uri{
27      /.*/:uri{{
          "http://www.w3.org/2000/01/rdf-schema#label":uri{
29          literal{ var VEHICLE_LBL }
          }
31      }}
      },
33    optional "http://triumph.example/schema/#depends-on":uri{
        desc (/.*/:uri &gt; "http://triumph.example/schema/#depends-on":uri)* /.*/:uri{{
35        "http://www.w3.org/2000/01/rdf-schema#label":uri{
            literal{ var DEP_LBL }
37        },
          "http://triumph.example/schema/#part-number":uri{
39          literal{ var DEP_NUM }
          }
41      }
      }}
43    }}
    }}
45 END
```

The query retrieves the human-readable label and part number of the part identified by the URI `http://triumph.example/part/0d92ie433` and optionally any dependent parts from the graph representation, utilizing the qualified descendant construct (see Bolzer (2005)) to recursively select the complete dependency tree. A simple HTML document is constructed using the selected information, demonstrating two major advantages of Xcerpt: the ability to construct arbitrary XML documents and the complete separation of query and construction, allowing independent modification of one or the other.

The same query against the set of triples is more complicated. The construction is the same as above, but in the query the transitive closure of the dependencies need to be collected separately using additional recursive rules. Two **CONSTRUCT** rules collect the information into ordered data terms labeled DEPENDENCY, each with two children: the dependent part as the first child and one of its dependencies as the second child. One rule collects all direct dependencies, the other recursively collects all indirect dependencies. The **GOAL** rule, the rule that constructs the final result, queries these DEPENDENCY data terms for any parts that the part identified by the URI `http://triumph.example/part/0d92ie433` depends on. Due to the backward-chaining nature of Xcerpt, only the dependencies of this part are actually collected, instead of the dependencies of all parts.

```
46   html{
      head{ title{ "Search Results" } },
48    body{
        h1{ &amp;join( var PART_LBL, "(", var PART_NUM, ")" },
50      p{ "Part for:" },
        ul{
52        all li{ var VEHICLE_LBL }
        },
54      h2{ "Dependencies:" },
        optional ul{
56        all li{
            &amp;join( var DEP_LBL, "(", var DEP_NUM, ")" )
58        }
        } with default "-"
60    }
    }
62 FROM
    and{
64    RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
```

```
                    "http://www.w3.org/2000/01/rdf-schema#label":uri{},
66                  literal{ var PART_LBL } ],
      RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
68                  "http://triumph.example/schema/#part-number":uri{},
                    literal{ var PART_NUM } ],
70    RDF-TRIPLE[ "http://triumph.example/part/0d92ie433":uri{},
                    "http://triumph.example/schema/#part-for":uri{},
72                  var VEHICLE ],
      RDF-TRIPLE[ var VEHICLE,
74                  "http://www.w3.org/2000/01/rdf-schema#label":uri{},
                    literal{ var VEHICLE_LBL } ],
76
      optional DEPENDENCY[ "http://triumph.example/part/0d92ie433":uri{}, var DEP ],
78    optional RDF-TRIPLE[ var DEP,
                    "http://www.w3.org/2000/01/rdf-schema#label":uri{},
80                  literal{ var DEP_LBL } ],
      optional RDF-TRIPLE[ var DEP,
82                  "http://triumph.example/schema/#part-number":uri{
              literal{ var DEP_NUM }
84          }
      }
86 END

88 CONSTRUCT
    DEPENDENCY[ var PART, var DEP ]
90 FROM
    RDF-TRIPLE[ var PART, "http://triumph.example/schema/#depends-on":uri{}, var DEP]
92 END

94 CONSTRUCT
    DEPENDENCY[ var PART, var DEP ]
96 FROM
    and[
98    RDF-TRIPLE[ var PART, "http://triumph.example/schema/#depends-on":uri{}, var X ],
      DEPENDENCY[ var X, var DEP]
100   ]
    END
```

The queries are dominated by long URIs, making them overly verbose. Even tough Xcerpt supports an abbreviation mechanism for XML namespaces, it is not suited for abbreviating the URIs used with RDF. Future research will look into a more generic abbreviation mechanism that will increase the readability of queries such as those shown in this thesis.

3.4 Inference Query: Finding Unknown Media Objects (Publishing)

The use cases introduces a multinational media conglomerate that manages a large, constantly updated database of media objects, such as books, movies and music tracks. As the conglomerate has rapidly expanded over the recent years, different parts of the database use different ontologies. To bridge the gap between these different parts, appropriate assertions using the vocabulary of RDF Schema have been added.

```
1 @prefix bmc:     &lt;http://big-media-conglomerate.example/ontology/#&gt; .
  @prefix dc:      &lt;http://purl.org/dc/elements/1.1/&gt; .
3
  []
5   baf:dollarPrice  "29.99" ;
    bmc:objectName   "J to the LO" ;
7   dc:author    &lt;http://big-media.example/author/1929/&gt; .
```

One of the conglomerate's editors needs to be informed about new titles that match certain criteria such as author, title and price range. For this purpose he creates a query that is regularly executed against the conglomerate's database and notifies the editor in form of web pages describing the matched items.

The ability to infer knowledge not explicitly included in a graph is one of the most important principles of RDF. This use case utilizes the subPropertyOf property from RDF Schema to specify relationships between properties, so that queries using a specific property match not only statements using the exact property but also for statements expressed using its subproperties. Two different approaches are available for incorporating inferencing into RDF queries using Xcerpt. One approach is to manually implement the desired inferencing on top of the raw RDF data that includes only explicitly asserted statements.

```
102    ns-default = "http://www.w3c.org/1999/xhtml"

104    html{
         head{ title{ &amp;join("Watched Media Item:", var TITLE) } },
106      body{
           h1{ "uery Match!" },
108        ul{
            li{ "Title: ", var TITLE },
110         li{ "Author: ", var AUTHOR },
            li{ "Price: USD", var PRICE }
112      }
       }
114 FROM
     and{
116    optional RDF{{
          var OBJNAME:uri{{
118          "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri{{
               desc ( /.*/:uri &gt; "http://www.w3.org/2000/01/rdf-schema#subPropertyOf":uri)*
120              "http://big-media-conglomerate.example/ontology/#objectName":uri{{}}
            }}
122      }}
       }},
124    RDF{{
         identity var ID /.*/:/.*/{{
126        or{
           "http://big-media-conglomerate.example/ontology/#objectName":uri{
128            literal{ var TITLE as /Money/ }
           },
130          var OBJNAME:uri{
             literal{ var TITLE as /Money/ }
132          },
         },
134        "http://purl.org/dc/elements/1.1/author":uri{
           literal{ var AUTHOR}
136        },
         "http://big-accounting-firm.example/scheme/1.0/#dollarPrice":uri{
138          var PRICE
         } where { var PRICE &lt; 30 }
140      }}
     }}
142  }
     END
```

In this example query, the URIs of properties that have objectName in their transitive closures over subPropertyOf predicates are bound to the variable OBJNAME. The query searches for media objects with a price less than 30 dollars and a title containing the word "Money". The title is checked against statements that have either objectName directly or one of its subproperties as property. The subproperties are specified as those containing the objectName property in their transitive closures over subPropertyOf statements and are bound to the variable OBJNAME. A simple XHTML page detailing the query result is constructed for each match.

89

The other approach to inferencing is to query the fully expanded RDF graph that contains all statements that are inferrable from explicitly asserted statements. By directing the query towards the graph built from triples augmented using the rules presented for RDFS above, the user needs only to query for the objectName property and not concern himself with the inferencing.

```
144   ns-default = "http://www.w3c.org/1999/xhtml"

146   html{
        head{ title{ &amp;join("Watched Media Item:", var TITLE) } },
148     body{
          h1{ "uery Match!" },
150       ul{
           li{ "Title: ", var TITLE },
152        li{ "Author: ", var AUTHOR },
           li{ "Price: USD", var PRICE }
154     }
      }
156 FROM
      "rdfs":RDF{{
158     identity var ID /.*/:/.*/{{
          "http://big-media-conglomerate.example/ontology/#objectName":uri{
160         literal{ var TITLE as /Money/ }
          },
162     },
        "http://purl.org/dc/elements/1.1/author":uri{
164       literal{ var AUTHOR}
        },
166     "http://big-accounting-firm.example/scheme/1.0/#dollarPrice":uri{
          var PRICE
168     } where { var PRICE &lt; 30 }
      }}
170 END
```

Instead of periodically evaluating the same query against a database that might or might not have changed since the last evaluation, a reactive system that queries changes in the database as they occur might be better suited for use case of this kind. XChange Bry *et al.* (2004*b*) is a reactive language for distributed event processing on the Web, based Xcerpt. A significant amount of the methods for querying RDF using Xcerpt is likely to be directly applicable to XChange and poses an interesting direction for future research.

3.5   Combination and Graph Merging: Customizing Content Delivery (Device Independence)

A mapping service provides mobile users with navigational maps that can be displayed on mobile phones. In the use case's scenario, a user requests a map of the surroundings of a race track he's headed to. In order to chose an image adequate for the user's phone, the mapping service dereferences an URI where a profile describing the capabilities of the user's phone is stored, merging it with a more specific profile the user has sent along with his request.

The device profile is described using the UAProf schema defined by the Open Mobile Alliance, an industry consortium of mobile phone makers. Profiles using this schema are already being distributed by most major mobile phone makers. Below is an excerpt of an actual profile, available at the URI `http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml` in RDF/XML format.

```
8   @prefix xsd:     <http://www.w3.org/2001/XMLSchema#>.
```

```
   []
10     prf:BitsPerPixel "8"^^xsd:int ;
       prf:ColorCapable "true"^^xsd:boolean ;
12     prf:CPU "Arm 7" ;
       prf:ImageCapable "true"^^xsd:boolean ;
14     prf:ScreenSize "101x80" ;
       prf:SoundOutputCapable "false"^^xsd:boolean ;
16     prf:Vendor "Panasonic" ;
       prf:Model "GD67" .
```

The mapping service has a large collection of maps in various sizes and numbers of colors, stored on the service's web servers. The maps are described by an RDF graph identified by the URI `http://example.com/maps.rdf`, including assertions about which maps depict a certain resource.

```
18 @prefix xsd: &lt;http://www.w3.org/2001/XMLSchema#&gt;.
   @prefix img: &lt;http://example.org/image#&gt;.
20
   &lt;http://example.org/racetrack&gt;
22   foaf:depiction &lt;http://example.org/racetrackmap.color.jpg&gt;;;
     foaf:depiction &lt;http://example.org/racetrackmap.bw.jpg&gt; .
24
   &lt;http://example.org/racetrackmap.bw.jpg&gt;
26   img:width "120"^^xsd:int ;
     img:height "100"^^xsd:int ;
28   img:hasColor "false"^^xsd:boolean.

30 &lt;http://example.org/racetrackmap.color.jpg&gt;
     img:width "100"^^xsd:int ;
32   img:height "75"^^xsd:int ;
     img:hasColor "true"^^xsd:boolean.
```

The user submits the following information in order to receive an image showing the desired map:

- The URI of his phone's default profile (`http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml`).

- An RDF graph describing his phone's current settings, overriding his phone's default profile. It is temporally stored at the location `file:///tmp/36e7ba4f.rdf`.

- The URI of the resource he is requesting a map of (`http://example.org/racetrack`). How the phone maps a real-world location to this URI is out of scope of this use case.

The use cases integrates RDF data from multiple sources, including locally stored data, data retrieved over the Internet and data provided on a query-by-query basis by the user. The query shown here implements the use case using six Xcerpt rules. The first three rules specify the RDF graphs to be queried: the user's phone's basic profile, a temporarily stored local file containing the user's override profile and the mapping service's map description database. The fourth and fifth rules extract the actual property-value pairs from the default and override profiles into DEFAULT-PROFILE and USER-PROFILE data terms, which are merged into PROFILE data terms in rule six, using a conditional query. The **GOAL** rule implements the actual query, querying the merged profile for the screen size and color capabilities of the user's phone, employing a regular expression to extract height and width of the screen from a literal containing both, and uses the data to search for a map that is adequate for it.

```
1   RDF-FILE[ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml", var Root ]
  FROM
3   in{
     resource{ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml" },
```

```
 5    var Root
      }
 7  END

 9  % Rule No. 2, load override profile
    CONSTRUCT
11    RDF-FILE[ "file:///tmp/36e7ba4f.rdf", var Root ]
    FROM
13    in{
       resource{ "file:///tmp/36e7ba4f.rdf" },
15     var Root
      }
17  END

19  % Rule No. 3, load database of maps
    CONSTRUCT
21    CONSTRUCT
      RDF-FILE[ "http://example.com/maps.rdf", var Root ]
23  FROM
      in{
25     resource{ "http://example.com/maps.rdf" },
       var Root
27    }
    END
29
    % Rule No. 4, abstract data out from default profile
31  CONSTRUCT
      DEFAULT-PROFILE[ var Prof,  var VALUE ]
33  FROM
      RDF{{
35      attributes{{
          origin{ "http://mobileinternet.panasonicbox.com/UAprof/GD67/04.xml" }
37      }},
        /.*/:/.*/{{
39        /http://www.openmobilealliance.org/tech/profiles/uaprof/.*#(var Prof as .*)/:uri{
           literal{ var VALUE }
41      }}
      }}
43  END

45  % Rule No. 5, abstract data out from override profile
    CONSTRUCT
47    USER-PROFILE[ var PROFILE, var VALUE ]
    CONSTRUCT
49      RDF{{
          attributes{{ origin{ "file:///tmp/36e7ba4f.rdf" } }},
51        /.*/:/.*/{{
            /http:\/\/www.openmobilealliance.org/tech/profiles/uaprof/.*#(var PROF as .*)/:uri{
53         literal{ var VALUE }
        }}
55      }}
      ]]
57  END

59  % Rule No. 6, merge default and override profiles
    CONSTRUCT
61    PROFILE[ var KEY, var VALUE ]
    FROM
63    if USER-PROFILE[ var KEY, var VALUE ] then
        USER-PROFILE[ var KEY, var VALUE ]
65    else
        DEFAULT-PROFILE[ var VENDOR, var MODEL, var KEY, var VALUE]
67  END

69  % Rule No. 7, use merged profiled data to query appropriate maps
    GOAL
71    var IMAGE_URI
    FROM[
73    and[
        PROFILE[ "ScreenSize", /(var SCREEN_W as [:digit:]+)x(var SCREEN_HT as [:digit:]+)/ ],
```

```
75    PROFILE[ "ColorCapable". var COLOR ],
      RDF{{
77      "http://example.org/racetrack":uri{{
          "http://xmlns.com/foaf/0.1/depiction":uri{
79          var IMAGE_URI:uri{{
              "http://example.org/image#width":uri{ literal{ var IMAGE_W } },
81            "http://example.org/image#height":uri{ literal{ var IMAGE_H } },
              "http://example.org/image#color":uri{ literal{ var COLOR } }
83          }} where { var IMAGE_W &lt;= var SCREEN_W and
                      var IMAGE_H &lt;= var SCREEN_H
85          }
          }
87      }}
      }}
89    ]
    END
```

3.6 Querying XML documents and their meta data

A publisher has accumulated a large collection of technical articles it supplies to numerous affiliated web sites. The company keeps track of the articles, using an RDF knowledge base that describes title, authors and subjects of each article.

```
34  @prefix dc: <http://xmlns.com/dc/elements/1.1/>.
    @prefix pub: <http://example.com/publishing/>.
36
    <http://exampletechupdate.com/article/how-to-sink-a-ship.html>
38    rdf:type pub:Article;
      dc:subject "Ship";
40    dc:title "How to sink a Pirate Ship";
      dc:creator _:author1 .
42
    _:author1
44    foaf:name "Peter Pan";
      foaf:make
46    <http://exampletechupdate.com/article/how-to-sink-a-ship.html> .
```

One day, the chief editor of the company receives a letter from a law firm, warning the company of inappropriate use of the trade mark KOLA in numerous of its articles, without naming any particular articles. The chief editor instructs his staff to quickly prepare a list of all articles making use of the trade mark sorted by author and citing the paragraphs containing the trademark.

The use cases demonstrates how a Xcerpt can be used to control a query against XML documents using meta data described using RDF. The query obtains meta data as well as the URIs of articles from the company's RDF knowledge base and uses the URIs as arguments to the resource specifications, in order to retrieve the articles over the Internet and search them for paragraphs that contain the allegedly infringing trademark. A HTML report in is constructed using the selected information.

```
      out{
172     resource{ "file:///home/editor/greylist.html" },
        html{
174       head{ title{ "Potentially Infringing Articles" } },
          body{
176         h1{ "Articles containing the word KOLA" },
            ul{
178           all li{
                var TITLE,
180             p{ "Author(s):", &join( all var AUTHOR ) },
                all blockquote{ var PARA }
```

```
182            }
             }
184        }
         }
186    }
    FROM
188    and{
        RDF{{
190        var URI:uri{{
            "http://www.w3.org/1999/02/22−rdf−syntax−ns#type":uri{
192             "http://example.com/publishing/Article":uri{{}}
            },
194          "http://purl.org/dc/elements/1.1/creator":uri{{
             /.*/:/.*/{{
196             "http://xmlns.com/foaf/0.1/name":uri{ literal{ var AUTHOR }  }
             }}
198          }},
             "http://purl.org/dc/elements/1.1/title":uri{ literal{ var TITLE } }
200       }}
        }},
202    in{
        resource{ var URI },
204    desc var PARA as p{ /.* KOLA .*/ }
        }
206    }
    END
```

Chapter 4

Conclusion and Outlook

In this article, a large number of use cases from W3C documents on XML and RDF querying have been considered: For each use case an implementation in Xcerpt has been provided and, in the XML case, compared to implementations in XQuery provided by the W3C. A number of essential observations can be drawn from the previous chapters:

- Nested Grouping: For both RDF and XML data grouping, e.g., of all e-mail addresses of one's friends, is a core functionality to be provided by any query language. Since the result of RDF and XML queries is however often structured data, nested groupings become essential, cf. Sections 2.2.2 and 2.5.3. Most navigational XML query languages such as XQuery use nested queries to express nested grouping. However, nested queries not only make queries hard to understand, they also make optimizing and reasoning (e.g., containment and typing) of queries more complex. A grouping construct such as Xcerpt's **all** eases the authoring and evaluation of such queries considerably.

- Rules: Xcerpt's rule mechanism has been shown both in the RDF and in the XML case to be a convenient mechanism to provide separation of concern and to allow transparent views of the data more better suited for particular tasks. For RDF data, a view mechanism seems to be essential for allowing users access to RDF data both in graph and triple form.

- Graph: Indeed, Xcerpt's graph data model and transparent handling of references has proven not only in the RDF case, but also in the XML case (e.g., Sections 2.5 and 2.9) to be more convenient than a strict hierarchical data model as XQuery uses. If efficient ways of evaluating queries against graph data can be developed, such a data model seems to be a better choice to achieve the versatility required by future Web data.

- Intertwined: Finally, the core thesis of this article has been detailed, namely that the same language can be used to provide convenient and easily understandable access mechanisms for both XML and RDF data.

Future work on use cases for Xcerpt will focus on expanding the work presented in this article in two directions:

- Already well underway is work on more elaborate usage scenarios that illustrate the vision underlying Xcerpt. These usage scenarios will soon be published in a companion report Badea *et al.* (2005).

95

· To further demonstrate the versatility of Xcerpt, the full RDF Data Access Use Cases will be implemented and, in the same style as the comparison with XQuery in the XML part, compared with SPARQL, the RDF query language in development at the W3C. For this more detailed description of the use cases are required. Also, accessing other forms of meta-data beyond RDF and RDFS using Xcerpt is being considered. Interesting candidates are, e.g., ISO's TopicMaps.

Bibliography

Alschuler, Liora, Robert H. Dolin, Sandy Boyer, and Calvin Beebe, 20008: *Clinical Document Architecture Framework.* Tech. rep., Health Level Seven (HL7).

Amer-Yahia, Sihem, Chavdar Botev, Stephen Buxton, *et al.*, 2005April: *XQuery 1.0 and XPath 2.0 Full-Text.* Tech. Rep. Working Draft, W3C.
URL `http://www.w3.org/TR/xquery-full-text/`

Apparao, Vidur, Steve Byrne, Mike Champion, *et al.*, 199810: *Document Object Model (DOM) Level 1 Specification.* Recommendation, W3C.
URL `http://www.w3.org/TR/REC-DOM-Level-1/`

Badea, Liviu, François Bry, Andreas Doms, *et al.*, 20053: *Development of Use Cases, Part II: Usage Scenarios for a Versatile Web Query Language.* Deliverable I4-D3, REWERSE.

Boag, Scott, Don Chamberlin, Mary F. Fernández, *et al.*, 20052: *XQuery 1.0: An XML Query Language.* Working draft, W3C.
URL `http://www.w3.org/TR/xquery/`

Bolzer, Oliver, 20052: *Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt.* Diplomarbeit/Master thesis, University of Munich.

Bry, François, Tim Furche, Liviu Badea, *et al.*, 20048*a*: *Identification of Design Principles.* Deliverable I4-D2, REWERSE.
URL `http://rewerse.net/publications.html`

Bry, François, Tim Furche, Liviu Badea, *et al.*, 2005: *Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages.* Journal of Semantic Web and Information Systems, **1**(2).

Bry, François, Paula-Lavinia P?trânjan, and Sebastian Schaffert, 2004*b*: *Xcerpt and XChange: Logic Programming Languages for Querying and Evolution on the Web.* In *Proc. Int. Conf. on Logic Programming*, LNCS. Springer-Verlag.
URL `http://rewerse.net/publications.html`

Buxton, Stephen and Michael Rys, 2003May: *XQuery and XPath Full-Text Requirements.* Working draft, W3C.
URL `http://www.w3.org/TR/xquery-full-text-requirements/`

Chamberlin, Don, Peter Frankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie, 2005*2*: *XML Query Use Cases*. Working draft, W3C.
URL http://www.w3.org/TR/xquery-use-cases/

Chamberlin, Don, Peter Frankhauser, Massimo Marchiori, and Jonathan Robie, 2001*2*: *XML Query Use Cases*. Working draft, W3C.
URL http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20010215

Clark, Kendall Grant, 2004*10*: *RDF Data Access Use Cases and Requirements*. Working draft, W3C.
URL http://www.w3.org/TR/rdf-dawg-uc/

Cowan, John and Richard Tobin, 2004*2*: *XML Information Set (2nd Ed.)*. Recommendation, W3C.
URL http://www.w3.org/TR/xml-infoset/

DeRose, Steve, Eve Maier, and David Orchard, 2001*June*: *Xml linking language (xlink) version 1.0*. Recommendation, W3C.
URL http://www.w3.org/TR/xlink/

Draper, Denise, Peter Frankhauser, Mary Fernández, *et al.*, 2005*2*: *XQuery 1.0 and XPath 2.0 Formal Semantics*. Working draft, W3C.
URL http://www.w3.org/TR/xquery-semantics/

Fernández, Mary, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh, 2005*2*: *XQuery 1.0 and XPath 2.0 Data Model*. Working draft, W3C.
URL http://www.w3.org/TR/xpath-datamodel/

Furche, Tim, François Bry, Sebastian Schaffert, *et al.*, 2004*8*: *Survey over Existing Query and Transformation Languages*. Deliverable I4-D1, REWERSE.
URL http://rewerse.net/publications.html

Hung, E., Y. Deng, and V. S. Subrahmanian, 2005: *RDF Aggregate Queries and Views*. In *Proc. Int. Conf. on Data Engineering*.

Katz, Howard, Don Chamberlin, Denise Draper, *et al.*, 2003*8*: *XQuery from the Experts: A Guide to the W3C XML Query Language*. First ed. Addison-Wesley.

Klyne, Graham, Jeremy J. Carroll, and Brian McBride, 2004*February*: *Resource description framework (rdf): Concepts and abstract syntax*. Recommendation, W3C.
URL http://www.w3.org/TR/rdf-concepts/

Malhotra, Ashok, Jim Melton, and Norman Walsh, 2005*2*: *XQuery 1.0 and XPath 2.0 Functions and Operators*. Working draft, W3C.
URL http://www.w3.org/TR/xpath-functions/

Marx, Maarten, 2004*6*: *Conditional XPath, the First Order Complete XPath Dialect*. In *Proc. ACM Symposium on Principles of Database Systems*, pp. 13–22. ACM.
URL http://turing.wins.uva.nl/~marx/pub/recent/pods04.pdf

Prud'hommeaux, Eric and Andy Seaborne, 2005*2*: *SPARQL Query Language for RDF*. Working draft, W3C.
URL http://www.w3.org/TR/rdf-sparql-query/

Robie, Jonathan, 2001: *The Syntactic Web*. In *Proc. XML Conference and Exhibition*.
URL `http://www.idealliance.org/papers/xml2001/papers/html/03-01-04.html`

Schaffert, Sebastian, 2004: *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich.
URL `http://www.pms.ifi.lmu.de/publikationen/`

Schaffert, Sebastian and François Bry, 20048: *Querying the Web Reconsidered: A Practical Introduction to Xcerpt*. In *Proc. Extreme Markup Languages*.
URL `http://rewerse.net/publications.html`

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

Walsh, Norman, 2003: *RDF Twig: accessing RDF graphs in XSLT*. In *Proc. Extreme Markup Languages*.
URL `http://www.mulberrytech.com/Extreme/Proceedings/xslfo-pdf/2003/Walsh01/EML2003Walsh01.pdf`

Walsh, Norman and Leonard Muellner, 199910: *DocBook: The Definitive Guide*. O?Reilly.
URL `http://www.oreilly.com/catalog/docbook/`

Zloof, Moshé M., 1975: *Query By Example*. In *AFIPS National Computer Conference*.

Zloof, Moshé M., 1977: *Query-by-Example: A Data Base Language*. IBM Systems Journal, **16**(4), pp. 324-343.
URL `http://www.research.ibm.com/journal/sj/164/ibmsj1604C.pdf`