JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 10, November-December 2004

On Query-Processing Issues for Non-Navigational Queries for XML

Won Kim, Cyber Databas Solutions, Inc., Austin, Texas
Wol Young Lee, **Hwan Seung Yong**, Department of Computer Science and Engineering, Ewha Institute of Science and Technology, Seoul, Korea

Abstract

In an earlier article, we motivated the need for structure-agnostic, that is, non-navigational, queries against XML documents. The conventional XML query languages require the users to know the structure of the XML documents and specify search conditions on the structure. However, the expressive flexibility of XML can give rise to many different representations and structures for the same document contents. A structure-agnostic query against XML documents is a very useful complement to the conventional navigation-based XML query languages. A structure-agnostic query language is very simple; however, the burden of processing a given query against XML documents of diverse representations and structures falls entirely on the query processor. In this article, we identify four key issues that arise in the automatic processing of structure-agnostic XML queries.

1 INTRODUCTION

In an earlier article [Kim, Lee, Yong 2004], we discussed a need for a query language for XML that does not require the user to specify the structure (paths) of the XML documents. The flexibility of XML gives rise to many different representations and structures for even the same document contents. In environments where different users may have created many different representations and structures for documents in XML, or where the precise structures of the documents are difficult to obtain, it may be highly useful for users to be able to specify only the element names and their values in query expressions, and have the query processor return matching documents or elements of the documents.

The fact that users are freed from having to know and specify the structures of the XML documents in query expressions means, however, that the burden of automatically navigating the hierarchical structures of the XML documents and matching the search conditions (element or attribute names and their values) against the elements or attributes and their values in the stored XML documents falls entirely on the query processor. We have defined a very simple structure-agnostic XML query language, called Chamois-XML-Query (CXquery), and designed and implemented a query processor, called Chamois-XML Query Processor, for evaluating queries expressed in the CXquery language. We have also measured and analyzed performance of the query processor.

In this article, we will only describe four key issues that arise in processing structure-agnostic queries, in order to provide concrete research issues for other researchers to pursue. The specific techniques and algorithms that we have developed and implemented in Chamois-XML Query Processor will be reported shortly elsewhere.

(To provide a running illustrative example, we take the following paragraph and two Figures from [Kim, Lee, and Yong 2004].) Suppose that we have XML documents on movies, as represented in Figure 1-1. Figure 1-2 (a) and (b) represent the XML document of the same content using hierarchical structures. If we are to search for the titles of movies whose genre is 'action', release year is '1994', and whose stars include 'Jean Reno', we would like to be able to state the search conditions simply as

genre = "action" and year = "1994" and actor = "Jean Reno"

regardless of the structure of the XML documents. Then a non-navigational content-based query (in XQuery) that includes the above search conditions would look something like

```
for $t in doc()//title
where genre = "action" and year = "1994" and actor = "Jean Reno"
return $t
```

The same structure-agnostic query would work for each of the alternate representations shown in Figure 1-2.

2 QUERY PROCESSING ISSUES

The Chamois-XML Query language can be formally defined as follows.

Expression ::= ("not")? Predicate (("and"| "or") ("not")? Predicate)*
Predicate ::= DataName ("=" | "<" | "<=" | ">" | ">=" | "!=" | "contains") DataValue
DataValue ::= numeric | string

where DataName is either an element name or an attribute name.

The processing of a CXquery consists of three steps.

- *Step (a)*: Evaluate each predicate (i.e., search condition) to find a potential matching XML document.
- *Step (b)*: Do the Boolean processing of all search conditions to find only the XML document that satisfies all search conditions.
- *Step (c):* For each matching document, process the result clause to return appropriate parts of the document.

```
<movie>
    <year>1994</year>
    <country>America</country>
    <country>France</country>
    <genre>drama</genre>
    <genre>action</genre>
    <title>Leon</title>
    <director>Luc Besson</director>
    <actor>Jean Reno</actor>
    <actor>Natalie Portman</actor>
</movie>
```

(a) Data described as elements

```
<movie>
    <general_info>
       <year>1994</year>
       <country>America</country>
       <country>France</country>
       <genre>drama</genre>
       <genre>action</genre>
    </general_info>
    <detail_info>
       <title>Leon</title>
       <people>
         <director>Luc Besson</director>
         <actors>
            <actor>Jean Reno</actor>
            <actor>Natalie Portman</actor>
         </actors>
       </people>
    </detail_info>
</movie>
```

(b) an arbitrary element name intervene between element names

```
<movie>
    <production year="1994"
          country="'America' 'France'">
    </production>
    <detail_info genre="'drama'  'action'"
                 title="Leon">
    </detail_info>
    <people actor= "Jean Reno"
           director="Luc Besson">
    </people>
</movie>
```

(c) Data described as attributes

```
<movie>
    <year yyyy="1994"></year>
    <country name1="America"
         name2="France"></country>
    <genre type1="drama"
            type2="action"></genre>
    <title name="Leon"></title>
    <director name="Luc Besson">
    </director>
    <actor name1= "Natalie Portman"
        name2= "Jean Reno "></actor>
</movie>
```

(d) attribute names intervene between an element name and its value

```
<movie>
    <year><yyyy>1994</yyyy></year>
    <country><name>America</name>
             <name>France</name></country>
    <genre><type>drama</type>
             <type>action</type></genre>
    <title><name>Leon</name></title>
    <director>
        <name>Luc Besson</name>
    </director>
    <actor><name>Jean Reno</name>
             <name>Natalie Portman</name></actor>
</movie>
```

(e) an element name intervene between an element name and its value

Figure 1-1 Example documents having a non-nested relationship among year, genre, and actor

```
<year>1994
 <country>America
  <genre>action
   <movie>
    <title>Leon</title>
    <director>Luc Besson</director>
    <actor>Jean Reno</actor>
   </movie>
    …
  </genre>
  …
 </country>
 …
</year>
```

(a) Data described as elements

```
<genre type="action">
 <country name="America">
  <year><yyyy>1994</yyyy>
   <movie>
    <title>Leon</title>
    <people  director="Luc Besson"
             actor="Jean Reno"
    </people>
   </movie>
    …
  </year>
   …
 </country>
…
</genre>
```

(b) Data described as attributes/ an arbitrary data intervene between an element name and its value

Figure 1-2 Example documents having a nested relationship among year, genre, and actor

Intuitively, Step (a) requires the query processor to navigate the XML document looking for the element or attribute name that appears in a search predicate in the query. Once the element or attribute name is found, the value associated with it is compared against the value in the search predicate. To illustrate this (so far very obvious observation), we represent an XML document in a hierarchical structure, shown in Figure 2-1. The rectangles represent elements or attributes included in elements, and the circles represent their values. We indicate an attribute included in an element in a dotted rectangle to the right of the element. Figure 2-1(a) shows how the search predicates genre=”action” and year=”1994” and actor=”Jean Reno” may be evaluated.

Unfortunately, the flexibility of XML, the structure-agnostic nature of CXquery, and the nature of the environments in which structure-agnostic queries may be used, all conspire to give rise to at least four key complications to automatic query processing. In the remainder of this section, we will discuss these issues.

First, in structure-agnostic queries, the users only need to specify search predicates. Each search predicate consists of the name of an element or attribute, an operator, and the value associated with the element or attribute. In an environment where the schema or DTD of XML documents is not precisely known or “fuzzy” (approximate) search is done, even the precise names of the elements and attributes may not be known. In such situations, the query processor needs to do similarity matching of the names that appear in the search predicates. The results of the query would then need to be ranked, as they are not necessarily all precise matches. For example, for the element names “actor”, “genre”, and “year”, the query processor may also need to search for names such as “performer”, “category”, and “date”, respectively.

Second, there may be intervening elements and/or an attribute between an element name and its corresponding value. An XML document of the same content as that represented in Figure 2-1(a) may easily be represented as in Figure 2-1(b). In Figure 2-1 (b), *type* intervenes between *genre* and “action”, *name* intervenes between *actor* and “Jean Reno”, and *yyyy* intervenes between *year* and “1994”. Furthermore, in Figure 2-1 (c), *genre, year,* and *actor* are represented as attributes, while in Figure 2-1 (d), *genre, year,* and *actor* are represented as elements but their values are represented as attribute values. In general, there may be an arbitrary number of elements, even a subtree of elements, between the element name and its corresponding value that match the search predicate in a query. There can also be an attribute name-- only one attribute -- between the element name and the corresponding value. As such, the query processor cannot assume that once an element whose name matches that in a search predicate is found, the child node of the element node is the data value of the element. This introduces significant implementation difficulties to the query processor for structure-agnostic query languages.

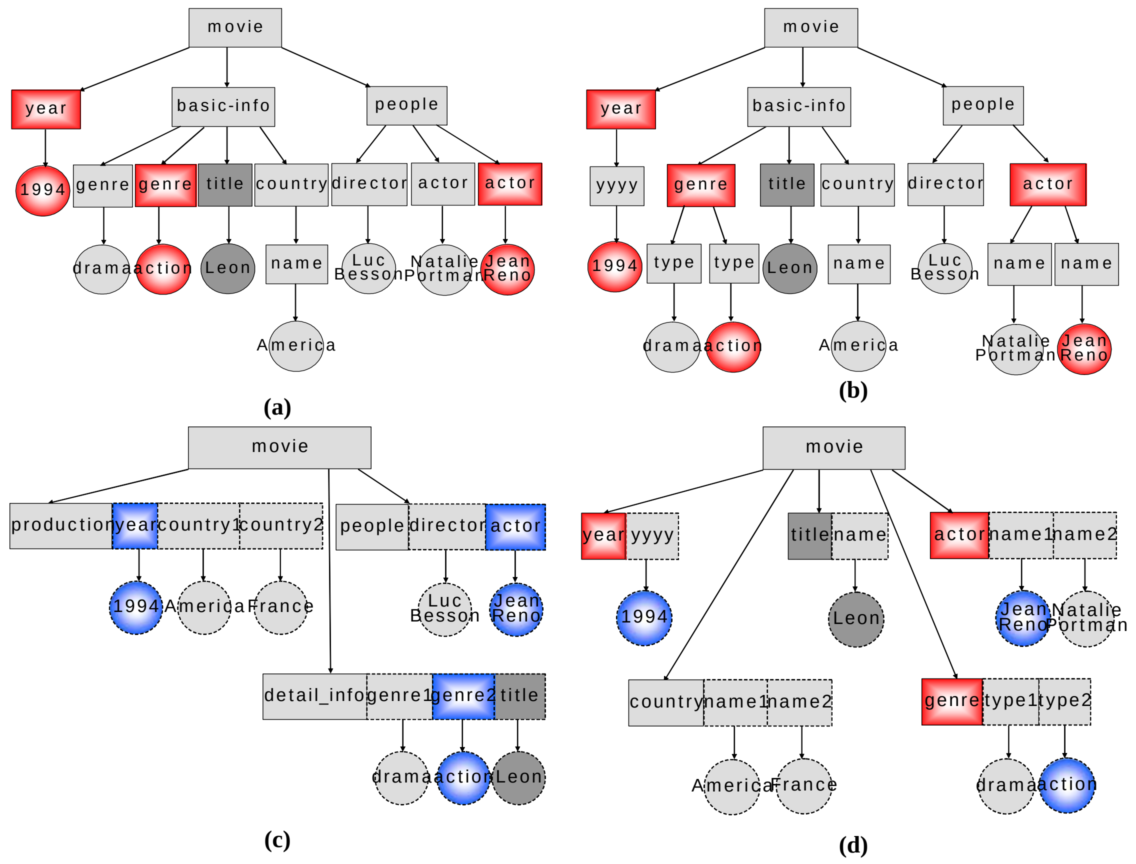

Figure 2-1 Representation Alternatives

Third, the fact that there may be intervening elements and/or an attribute between an element and its corresponding value, as observed above, leads to "semantic uncertainty" in the association between the element and the value. Let us consider the *actor* and "Jean Reno" pair. Suppose that between *actor* and "Jean Reno", there is an intervening name *family* as follows: (There may in reality be an arbitrary number of intervening names.)

```
<actor>
    <family>
        <name>Jean Reno</name>
        ...
    </family>
</actor>
```

In this example, "Jean Reno" is the value associated with the element or attribute "family" of "actor". However, if the query processor is to blindly navigate from *actor* and, upon finding "Jean Reno", declare that the search predicate "*actor* = *Jean Reno*" is true, the semantic correctness of the result may be in question. This issue of semantic correctness may arise even if there is no intervening element or attribute, since by *actor* the user may have in mind a "theater actor" rather than a "movie actor". In the following, both the theater actor and the movie actor are matched.

```
<actors>
    <theater>
        <actor>Jean Reno</actor>
    </theater>
    <movie>
        <actor>Jean Reno</actor>
    </movie>
</actors>
```

In any case, in environments where structure-agnostic queries are used, the semantic correctness of the query result is in general in some doubt, and the query processor must employ some means to attach a "confidence estimate" to each query result that it returns.

Fourth, for the purpose of query-processing optimization and of preventing erroneous results, it is necessary to compute the nearest common ancestor (NCA) of all element and attribute names that appear in the search predicates in the query. We note that the problem of determining the NCA arises even in the processing of path-based XML query languages, such as XQuery. However, the problem is more difficult in the case of CXquery, since the structure of the XML hierarchy is not specified in CXquery. In Figure 2-2 (a), the *year* element is the NCA of *year, actor*, and *genre* as *actor* and *genre* are nested in *year*. If the location of *genre* and *year* is opposite, the result is *genre*. In Figure 2-2 (b), the NCA of *genre, year,* and *actor* is a new, higher-level node (the blank rectangle). There are two reasons for computing the NCA. One is to limit the scope of navigation of the XML hierarchy, both when doing Step (a) and Step (c) of query processing. Suppose that there are higher nodes to the "root" node in Figure 2-2. In general, the query processor needs to start its search for the desired element or attribute names from the root node of the XML tree, and possibly traverse the tree multiple times. Once the query processor can determine the NCA, it can confine its subsequent traversal of the tree to the subtree rooted at the NCA. For example, in Figure 2-2 (a), if the query processor has determined that the NCA of *genre* and *actor* is *year*, in doing the result clause processing, it can return *year* as the result of the AND-ing of *genre*, *actor*, and *year* without subsequent traversal of the entire tree. Another reason is to prevent the tree traversal of a single subtree from "over-flowing" into another subtree. For example, suppose the root of an XML tree is named *movies*, and its child node is named *movie* as follows.

```
<movies>
   <movie>
      <general_info>
         <year>1994</year>
         <genre>action</genre>
      </general_info>
      <detail_info>
         <actors>
            <actor>Jean Reno</actor>
         </actors>
      </detail_info>
   </movie>
   <movie>
      <general_info>
         <year>1994</year>
         <genre>action</genre> ...
   </movie>
</movies>
```

Then the query evaluation involving *movies* should be confined to within each *movie* document instance, and, for example, the search predicate on *actor* in one *movie* document should not be Boolean-combined with the search predicate on *genre* in another *movie* document. Further, the result clause of a matching *movie* document should not be computed against another *movie* document.

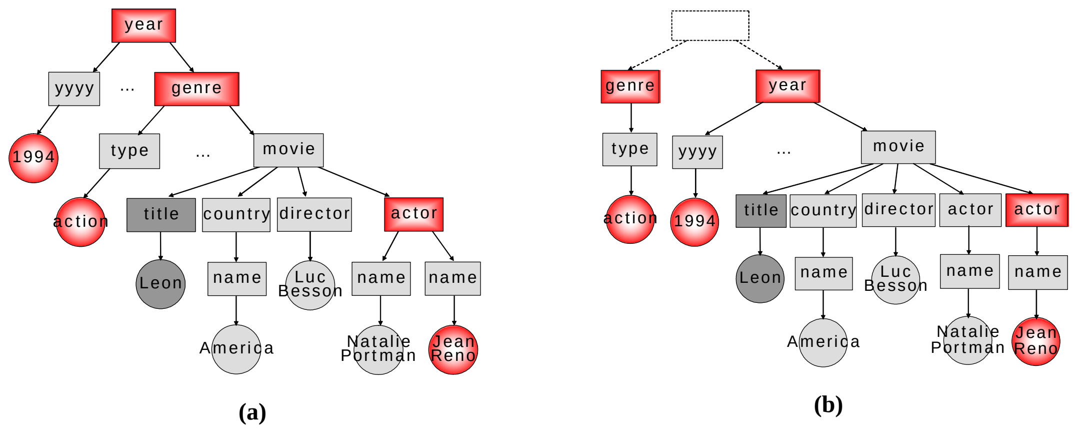

Figure 2-2 Determining the Nearest Common Ancestor

3   CONCLUDING REMARKS

In this article, we identified four key issues that must be addressed in the query processor for structure-agnostic XML queries, that is, queries that are formulated with only the search conditions but not the search paths along the hierarchical representations of XML documents. The purpose of the article is to encourage researchers in XML query languages and XML query processing techniques to take on some of these issues in order to improve on the techniques and algorithms that we have designed and implemented. The details of our techniques and algorithms will be reported shortly elsewhere.

REFERENCES

[Kim, Lee, and Yong 2004] Won Kim, Wol Young Lee, Hwan Seung Yong. "On Supporting Structure-Agnostic Queries for XML", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 27-35. http://www.jot.fm/issues/issue_2004_07/column3

About the authors

Won Kim is President and CEO of Cyber Database Solutions (http://www.cyberdb.com/) in Austin, Texas, USA. He is Editor-in-Chief of ACM Transactions on Internet Technology (http://www.acm.org/toit), and Chair of ACM Special Interest Group on Knowledge Discovery and Data Mining (http://www.acm.org/sigkdd). He is the recipient of the ACM 2001 Distinguished Service Award.

Wol-Young Lee is a Ph.D. candidate in the Department of Computer Science and Engineering in Ewha Women's University, Seoul, Korea. Her research interests include XML, database systems, and programming languages. Her email is wylee at ewha.ac.kr.

Hwan-Seung Yong is an associate professor with the Department of Computer Science and Engineering in Ewha Women's University, Seoul, Korea. His research interests include multimedia database systems, data mining and bioinformatics, and Internet computing. He received a Ph.D. degree in computer engineering from Seoul National University. His email is hsyong at ewha.ac.kr.