XPath Extension for Querying Concurrent XML Markup*

Ionut E. Iacob Alex Dekhtyar Wenzhong Zhao

University of Kentucky
Dept. of Computer Science
773 Anderson Hall
Lexington, KY 40506-0046, USA
{eiaco0,dekhtyar,wzhao0}@cs.uky.edu

Abstract

XPath is a language for addressing parts of an XML document. It is used in many XML query languages and it can be used by itself for querying XML documents. While XPath is, in general, efficient for querying individual XML documents, it lacks the features for querying over collections of documents or joining parts of the same document.

As the amount of complex document-centric XML data is continually increasing, querying such documents has drawn surprisingly little attention. We propose an XPath axes extension to deal with querying collections of document-centric XML documents sharing the same content (called concurrent XML). The algorithms we propose to evaluate the extended axes work in linear time combined complexity (number of documents and total size of documents).

1 Introduction

XML, having initially been designed for large scale text publishing, has rapidly evolved as a standard for a wide variety of data exchange and representation tasks. With the increase in the volume of data in XML format, storing and querying XML has been the subject of intensive research [14, 10, 25, 8]. Currently, all major commercial relational databases offer some form of XML storage and query support. Two major kinds of XML documents emerge from applications: *data-centric* and *document-centric*. Data-centric documents are characterized by a fairly regular structure and occur as a standard format for structured data exchange and representation of semistructured data. Document-centric XML has, in general, a much more irregular structure and is often encountered as the means of document markup. In recent years, a number of applications of XML to document encoding lead to markup, that could not be stored in a single well-formed XML document. In this paper we address the problem of querying such complex document-centric XML markup by extending the XPath syntax and semantics.

XPath [5] is a language for addressing parts of an XML document. It is used as an XML query language by itself or as a core part of other XML query languages (e.g., XQuery [8]) or other XML related technologies (e.g., XPointer [7], XSLT [6]). XPath support is a common feature today for XML databases. However, XPath lacks an important feature for a query language: addressing parts of XML documents over a collection of documents or joining interrelated parts in the same document. Complex document-centric XML encodings (for instance, electronic editions of literary texts, especially those of old manuscripts [20]) deal frequently with a high density of markup. Moreover, the diverse information to be encoded contradicts a fundamental constraint of an XML document: well-formedness. Markup that contradicts the well-formedness of an XML document is called *concurrent markup* [29, 26]. The most common example of such conflicting, or overlapping concurrent markup is encoding of individual physical lines of the manuscript versus the sentence structure of the text: physical lines often end at mid-sentence, sometimes even at mid-word, causing the scopes of the XML elements representing sentences and physical lines to overlap[21]. This problem had been formulated some time ago by humanities researches (see, for instance, [24]) and a variety of approaches have been proposed since then ([12, 9, 26, 13, 27, 19]). The work above, for the most part, concentrated on providing the human editor with the means of creating well-formed XML documents that encoded all necessary features, or, alternatively, with representing such markup using a non-XML approach. To the best of our knowledge, the problem of efficiently accessing data stored in concurrent XML hierarchies have not been yet addressed.

*Technical Report #TR 394-04, Department of Computer Science, University of Kentucky, Lexington, KY 40506

Concurrent XML markup often occurs during preparation of image-based electronic editions of manuscripts[21]: the scopes of different marked-up features overlap in many situations. Consider, for example, the image of a manuscript folio[1] shown in Figure 1. Given such an image, an editor[1] wants to mark up in the manuscript content transcription the physical lines (transcript markup) together with the verse lines (edition markup) of the poetry. The fragment of the text found on the image fragment (part of it is damaged and cannot be seen, though) is shown below the image in Figure 1, with the overlapping physical and verse line markup. Other potential conflicts in the editorial markup of this manuscript appear from marking up words, damaged areas, or text restorations (as can be easily deduced from Figure 1, the text content of image folio can be only partially recovered and editorial restorations often occur in such situations)[21].

Solutions dealing with concurrent markup were proposed in [12, 9, 26, 13, 19]. Two straightforward approaches are *fragmentation* (nodes are fragmented to preserve well-formedness, each fragment is marked with a special "glue" attribute with the same value) and *milestone elements* (empty elements) [26]. For the instance of concurrent markup in Figure 1 *fragmentation* and *milestone elements* would lead to XML fragments shown in Figure 2, top and bottom respectively.

However, a simple information request like "Get the textual content of a <vline> node" poses some challenges for both approaches. For *fragmentation*, this requires a join of two fragments. For *milestone elements* (where the "content" of a milestone element is the text between two consecutive milestones), answering the query requires multiple joins of all text nodes between the two consecutive milestones. These require not only special post-processing of the query results but also a unnatural query formulation for even simple queries like retrieving the content of a node. We also note that a number of approaches cited above ([12, 19]) suggest going beyond the syntax of XML, while another apporach, suggested by Durusau and O'Donnel[13], presents, essentially, an inverted index (each "atomic" unit of content is encoded with a series of XPath expressions for each of the concurrent hierarchies) embedded inside an XML document, and is hard to process.

In [9] we have formalized the notions of a concurrent XML hierarchy (CMH) and a distributed XML document, representing the concurrent markup. Our approach is to treat concurrent XML encoding as a single *virtual* document. [9] addressed the problem of efficient, information-preserving transformations between distributed XML documents and regular XML documents. Our work here continues the work started in [9] by describing how XPath can be extended to traverse and query distributed XML documents efficiently.

We can summarize the contributions of this paper as follows:

- An extension for XPath language for traversing/querying multiple components in a concurrent XML hierarchy. The representation we use for the concurrent XML hierarchy is a novel approach for reducing the complexity of schemas over document centric XML by using a set of simpler schemas over a collection of documents.
- Algorithms for efficient (linear time) evaluation of extended XPath axes.
- Experimental results to support the efficiency of the algorithms we propose.

Our proposed extension to XPath includes new semantics for six XPath axes (*ancestor, descendant, ancestor-or-self, descendant-or-self, following, preceding*) and five new axes (*following-overlapping, preceding-overlapping, overlapping, ancestor-or-overlapping, descendant-or-overlapping*) dealing with overlapping markup. We show that the extended semantics of the six regular XPath axes *specializes* to the standard XPath semantics when the XML document does not contain markup conflicts.

The rest of the paper is organized as follows. In Section 2 we give formal definitions for the concurrent XML hierarchy, its data structure representation and a motivating example. In Section 3 we introduce the XPath language and the XPath axes extensions we propose. The algorithms for efficient evaluation of the extended XPath axes are given in Section 4 and the experimental results for testing the algorithms implementations are given in Section 5. We present the related work in Section 6 and conclude in Section 7.

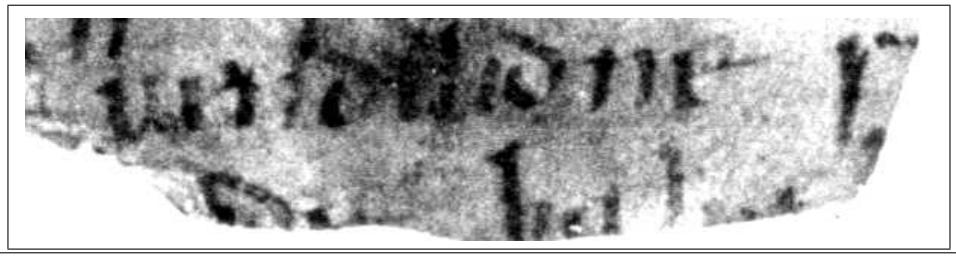
2 Concurrent XML Hierarchies

2.1 Background

A concurrent XML hierarchy as defined in [9] is a collection of DTDs sharing the same root element. Using a *concurrent markup hierarchy* (CMH), a large, complex schema can be broken down into a number of smaller, less complex schemas. However, the most important benefit of using CMHs is the ability to define and use logical hierarchies XML elements with no conflicts between markup tags inside of the same hierarchy.

Before we give formal definitions of CMHs we introduce some notation used throughout the paper. First, all XML document instances in this paper (which we refer to as *documents*) are considered to be well-formed, unless explicitly specified otherwise. For a DTD T we let $elements(T)$ be the set of *element type* names as they appear in Element Type Declarations in T[3]. For a document d we let $elements(d)$ be the

[1]The humanities researcher who prepares an electronic edition



"<line>gesceaftum unawendendne sin</line><line>gallice sibbe gecynde þa</line>"

"gesceaftum unawendendne <vline>singallice sibbe gecynde </vline>þa"

Figure 1: An image[2] fragment of King Alfred's Boethius manuscript [1], folio 036v, and concurrent XML markup.

"<line>gesceaftum unawendendne
<vline linkID='1'>sin</vline></line><line><vline linkID='1'>gallice sibbe gecynde
</vline>þa</line>"

"<line>gesceaftum unawendendne <vline/>sin</line><line>gallice sibbe gecynde <vline/>þa</line>"

Figure 2: Fragmentation and milestone elements

set *element types* (*tag names*) in d [3]. It follows that, a necessary condition for a document d to be valid[3] w.r.t. some DTD T is $elements(d) \subseteq elements(T)$. A tree representation used by XPath [5] to model an XML document has seven types of nodes, is rooted at the *root element* node and all *text nodes* are terminal nodes (leaves). We let $nodes(d)$ represent all *element nodes*, *text nodes* and the *root node* in the document d, as they are defined in the XML tree model employed by XPath [5]. For a document d we let $tree{-}nodes(d)$ be the set of nodes in XPath tree representation (note that $nodes(d) \subseteq tree{-}nodes(d)$ as the former takes into account only some of the nodes) and let $text{-}nodes(d)$ be all *text nodes* in d (hence $text{-}nodes(d) \subseteq nodes(d)$).

A total order is defined over all nodes in a document d as the *pre-order* traversal of the document tree model. For two nodes $x, y \in nodes(d)$ we denote by $x < y$, or equivalently $y > x$, whenever node x precedes node y in document order of d. For any node $x \in nodes(d)$ we let $string{-}value(x)$ be the concatenation of the text-values of all text node descendants of x in document order [3] (if x is a *text node* then $string{-}value(x)$ is the textual value of x). For a document d we let $root(d) \in nodes(d)$ denote the *root element* of d and we let $string{-}value(d) = string{-}value(root(d))$.

For a document d we define functions $start, end$ (which describe the position of a node relative to the document textual content) as
$start, end : nodes(d) \rightarrow \{0, 1, \ldots, |string{-}value(d)|\}$
where, $\forall x \in nodes(d)$:
• $start(t)$ is the character position in $string{-}value(d)$ where $string{-}value(t)$ begins; if $string{-}value(t) = \epsilon$, then $start(t) = start(p)$ where $p \in nodes(d)$ is the first node (in reverse document order) that precedes t such that $string{-}value(p) \neq \epsilon$ or $start(t) = 0$ if no such node p precedes t;
• $end(t)$ is the character position in $string{-}value(d)$ before which $string{-}value(t)$ ends; if $string{-}value(t) = \epsilon$, then $end(t) = end(f)$ where $f \in nodes(d)$ is the first node (in document order) that follows t such that $string{-}value(f) \neq \epsilon$ or $end(t) = |string{-}value(d)|$ if no such node f follows t.
For instance, $start(root(d)) = 0$ and $end(root(d)) = |string{-}value(d)|$.

Definition 1 *In a DTD T, $y \in elements(T)$ is an* ancestor *of $x \in elements(T)$ if one of the following holds:*
(i) x is listed in y's Element Type Declaration.
(ii) Some element z listed in y's Element Type Declaration is an ancestor *of x.*

Definition 2 *[9] A* concurrent markup hierarchy *CMH is a tuple $CMH = \langle \rho, \{T_1, T_2, ..., T_k\} \rangle$ where:*
• *ρ is an XML element called the* root *of the hierarchy;*
• *T_i, $i = \overline{1,k}$ are DTDs such that:*
(i) $\forall 1 \leq j \leq k, i \neq j$, $elements(T_i) \cap elements(T_j) = \{\rho\}$;
(ii) $\forall t \in elements(T_i) - \{\rho\}$, ρ is an ancestor of t in T_i.

An example of concurrent markup hierarchy is shown in figure Figure 3 (top box).

Definition 3 *[9] A* distributed XML document *D over a concurrent markup hierarchy CMH =*

[2]Digitized by Kevin Kiernan (University of Kentucky) and used with permission of the British Library Board.

$\langle \rho, \{T_1, T_2, ..., T_k\}\rangle$ *is a collection of XML documents:* $D = \langle d_1, \ldots, d_k\rangle$ *where (i)* $(\forall 1 \leq i \leq k)$ d_i *is valid w.r.t.* T_i*; (ii)* $string{-}value(d_1) = string{-}value(d_2) = ... = string{-}value(d_k)$*, and (iii)* $root(d_1) = root(d_2) = ... = root(d_k) = \rho$.

We say that for a distributed document D, $string{-}value(D) = string{-}value(d_1)$ *and* $root(D) = root(d_1)$.

A *distributed XML document* (see Figure 3, bottom box) allows us to distribute conflicting markup into separate documents. However, D is not an XML document itself, rather it is a *virtual* union of the markup contained in d_1,...,d_k. The problem of creating XML document instances that incorporate all information in a *distributed document* has been addressed in [9]. The focus of this paper is on extending XPath syntax and semantics for addressing/traversing parts of a distributed XML document.

Our next step is to define the abstract data model for distributed XML documents, which plays the same role as DOM trees do for regular XML. For a distributed XML document $D = \langle d_1, \ldots, d_k\rangle$, we will use set notation $d_i \in D$ to specify that d_i is a component document of D. Similarly, we will slightly abuse notation and write $D - d$ to represent a distributed XML document that consists of all components of D except for d. We also let $nodes(D)$ denote the set $\cup_{i=1}^{k} nodes(d_i)$. Given a node $x \in nodes(D)$, we let $doc_D(x)$ denote the document $d \in D$, such that $x \in nodes(d)$. Given a string s, we denote by $|s|$ the *length* of the string (number of characters in s). We also let $substring(s, i_1, i_2)$ denote the substring of s from position i_1 *up to but not including position* i_2 (here positions start from 0 up to position $|s|-1$), and we let ϵ denote the *empty string*.

For representing a distributed XML document we use a *General Ordered-Descendant Directed Acyclic Graph (GODDAG)* data structure proposed in [27]. Informally, a GODDAG for a distributed XML document $D = \langle d_1, \ldots, d_k\rangle$ can be thought of as the graph that unites the DOM trees of individual components of D, by merging the root node and the text nodes. However, because of possible overlap in the scopes of XML elements from different component documents, GODDAGs will feature one more node type, that we call here *leaf node*, not found in DOM trees. In a GODDAG, leaf nodes are children of the text nodes, and they represent a consecutive sequence of content characters that is *not broken by an XML tag in* **any** of the components of the distributed XML document. While each CMH component will have its own text nodes in a GODDAG, the leaf nodes will be shared among all of them. Given a distributed XML document $D = \langle d_1, \ldots, d_k\rangle$, we can compute the set of leaf nodes using the following algorithm:

for each $d \in D$
  for each $t \in text{-}nodes(d)$
    $i = start(t)$
    while $i < end(t)$
      $m = min\{j |\ j > i \wedge \exists d \in D$
            $\exists x \in text{-}nodes(d)$
            $(j = start(x) \vee j = end(x))\}$
      create *leaf node* parented by t and
          with textual content $substring(S, i, m)$
      $i = m$

In other words, *leaf nodes* are obtained by projecting each *start tag* and *end tag* from all component documents of D on the $string{-}content(D)$, at corresponding positions, then taking largest contiguous sequences of content characters not separated by markup to be the scope of individual leaf nodes. For a distributed document D we let $leaf{-}nodes(D)$ represent the set of all *leaf nodes* in D and we extend the domain of functions $string{-}value, start$, and end over the $leaf{-}nodes(D)$ set. For *leaf nodes* these functions are defined in the same way as for *text nodes*. We define two new functions, $first{-}leaf, last{-}leaf : nodes(D) \rightarrow leaf{-}nodes(D)$. Given an element, or text node x, these functions return the leftmost and the rightmost (respectively) leaf nodes in the subtree of x. If $string{-}value(x) = \epsilon$, then $first{-}leaf(x), last{-}leaf(x)$ return the first following (respectively the first preceding), in reverse document order, *leaf node* for x (or NIL if such nodes do not exist). We enumerate below some useful properties of leaf nodes.

Proposition 1 *Let* $D = \langle d_1, \ldots, d_k\rangle$ *be a distributed XML document.*

- *If* $l \in leaf{-}nodes(D)$ *then* $|string{-}value(l)| > 0$.
- $string{-}value(D)$ *is the concatenation of all* $string{-}value(l)$, $l \in leaf{-}nodes(D)$ *where the leaves* l *are taken in document order.*
- $\forall d \in \{d_1, \ldots, d_k\}$, *if* $l \in leaf{-}nodes(D)$ *then* $\exists t \in text{-}nodes(d)$ *such that* $start(t) \leq start(l) < end(l) \leq end(t)$.
- $\forall d \in \{d_1, \ldots, d_k\}$, *if* $t \in text{-}nodes(d)$ *then* $\exists l \in leaf{-}nodes(D)$ *such that* $start(t) \leq start(l) < end(l) \leq end(t)$.

Definition 4 *Let* $D = \langle d_1, \ldots, d_k\rangle$ *be a distributed XML document. A GODDAG of* D *is a directed acyclic graph* (N, E) *where the sets of nodes* N *and edges* E *are defined as follows:*

- $N = \cup_{i=1}^{k}(tree{-}nodes(d_i) - \{root(d_i)\})$
  $\cup\, leaf{-}nodes(D) \cup \{r\}$
- $E = \cup_{i=1}^{k}\{(r, x) | x \in tree{-}nodes(d_i) \wedge$
  $root(d_i)$ *is the parent of* $x\} \cup$
  $\cup_{i=1}^{k}\{(x, y) | x, y \in tree{-}nodes(d_i) - root(d_i) \wedge$
  x *is the parent of* $y\} \cup$
  $\cup_{i=1}^{k}\{(x, y) | x \in text{-}nodes(d_i),$
  $y \in leaf{-}nodes(D) \wedge$
  $start(x) \leq start(y) < end(y) \leq end(x)\}$

A GODDAG of D, basically, joins at the root level and leaf level, of all tree models (DOM trees) of doc-

T_1 = <!ELEMENT r (line)+>
<!ELEMENT line (#PCDATA)>

T_2 = <!ELEMENT r (vline)+>
<!ELEMENT vline (w)+>
<!ELEMENT w (#PCDATA)>

T_3 = <!ELEMENT r (#PCDATA|res)*>
<!ELEMENT res (#PCDATA)>

T_4 = <!ELEMENT r (#PCDATA|dmg)*>
<!ELEMENT dmg (#PCDATA)>

$\mathbf{CMH} = \langle \mathbf{r}, \{\mathbf{T_1}, \ldots, \mathbf{T_4}\} \rangle$

d_1 = "<r><line>gesceaftum unawendendne sin</line><line>gallice sibbe gecynde þa</line></r>"

d_2 = "<r><vline><w>gesceaftum</w><w>unawendendne</w></vline><vline><w>singallice</w><w>sibbe</w><w> gecynde</w></vline><vline><w>þa</w></vline></r>"

d_3 = "<r><res>gesceaftum una</res>wendendne s<res>in</res><res>gallice sibbe gecyn</res>de þa</r>"

d_4 = "<r>gesceaftum una<dmg>w</dmg>endendne singallice sibbe gecyn<dmg>de þa</dmg></r>"

$string{-}value(d_1) = .. = string{-}value(d_4)$ = "gesceaftum unawendendne singallice sibbe gecynde þa"

$\mathbf{D} = <\mathbf{d_1}, \ldots, \mathbf{d_4}>$

Figure 3: A concurrent XML hierarchy **CMH**, and a distributed XML document **D**.

uments in D. Consequently, each node in $nodes(D)$ has $root(D)$ as an ancestor, and each leaf node in $leaf{-}nodes(D)$ has exactly k parents, one for each document in D. Hence, for a leaf $l \in leaf{-}nodes(D)$ we denote as $parent(d_i, l), 1 \leq i \leq k$, the parent of leaf l in $nodes(d_i)$.

The GODDAG of the distributed XML document in Figure 3 is given in Figure 4. Each node of the GODDAG in Figure 4 has a label (a number appended to the node name), solely for the purpose of ease of identification. *Leaf nodes* are represented as bounding boxes around the content substrings and are labelled with numbers 1, 2,...,11. We identify them as "l1",...,"l11". All other nodes are represented as circles. Text nodes are labelled t1, t2, ..., t17, other nodes are labelled by their *node name* and a number (to make distinction between multiple occurrences of the same node name). In order to make the figure clear, we draw the *root node* twice, at the top and bottom of the figure.

2.2 Querying Concurrent XML

Concurrent markup hierarchies have been observed in many applications, such as, for example, building image-based electronic editions for historic manuscripts[21]. In this section, we discuss a number of possible approaches to querying distributed XML documents.

Our example, represented in Figure 3 and Figure 4 as a CMH and distributed XML document and its GODDAG, comes from the Electronic Boethius, a project at the University of Kentucky, devoted to creation of an image-based Electronic Edition of King Alfred's Boethius manuscript. The markup hierarchies used in the example are somewhat simplified, but represent real features of the manuscript that are encoded by the editor. The four hierarchies define physical organization of the document by lines, structure of the text (verse lines and individual words), text restorations and manuscript damage. The text used in the example represents approximately the content of the fragment shown in Figure 1.

Even with such simple encoding, the editor may want the Electronic Edition software to be able to display the following information:

1. Find all damaged characters.
2. Find all words containing damaged characters.
3. Find all words containing damaged characters ONLY.
4. Find all damaged characters which have been restored from other manuscripts.
5. Find all words containing damaged characters which have been restored from other manuscripts.

The first question can be answered by constructing an XPath query //dmg, involving elements from only one hierarchy, d_4. This query can be dealt with by an XPath processor in a straightforward manner. In order to answer the next four questions, however, we have to construct XPath queries involving elements from different hierarchies. There are two "simple" approaches to querying across different hierarchies:

a) Issue multiple standard XPath queries, one to each hierarchy, and then construct the final result by merging respective outputs together;

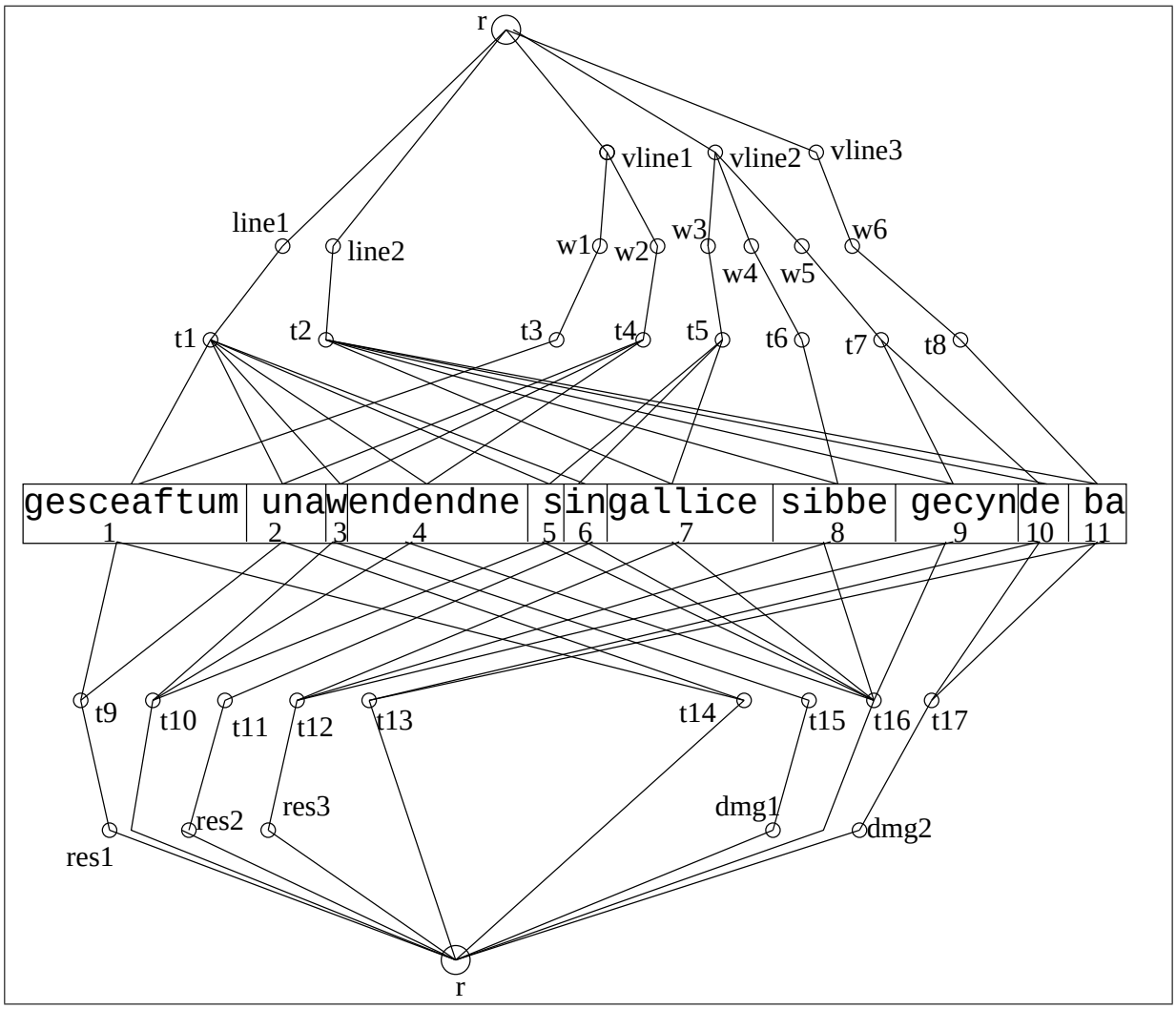


Figure 4: A GODDAG for the distributed document **D** in figure 3

b) Merge different hierarchies into a single master XML document that incorporates all the markup (following [9]), and then use queries involving only standard XPath expressions to query the master document.

Let us now examine what each of the approaches would do in order to formulate and process queries 2-4 from the list above. First let us take a look at method a). Using this approach, an concurrent XML XPath processor will be a mediator, that, given a query, (a) determines the hierarchies involved, (b) issues XPath query for each such hierarchy, (c) invokes standard XPath processor on each query, (d) receives the data back from the XPath processor and, finally, (e) merges the results. While, such an architecture is popular for a variety of heterogenous data management and XML integration systems [22, 23], let us look at how it would work on concurrent hierarchies. Consider, for example, question 5 from the list above. First thing we note, is that some formal syntax for representing query over concurrent hierarchies is needed. Assuming that the mediator has accepted this query in some such format, parsed it and understood, on its next step, it will determine that three hierarchies, d_2, d_3 and d_4 are involved in the query, and will construct the XPath expressions for each of the three hierarchies that are shown in Table 1.

After each XPath query is evaluated by a XPath processor, the results (also shown in Table 1) will be sent back to the mediator. We notice that ***xpath1*** returns all the words in the specified document. When the results of the three queries are merged, a lot of the information returned by ***xpath1*** will be discarded. This means that, on one hand, the mediator must accept a significant amount of responsibility for building the final answer set, while at the same time, processing independent XPath queries posed to individual hierarchies, may be a waste of time and effort (especially in large documents). Also, the process of merging the XPath query results has to be, by necessity,

QueryID	XPath Expression	Answer Set
xpath1:	document(d_2)//w	{<w>gesceaftum</w>, <w>unawendendne</w>, <w>singallice</w>, <w>sibbe</w>, <w> gecynde</w>, <w>þa</w>}
xpath2:	document(d_3)//res	{<res>gesceaftum una</res>, <res>in</res>, <res>gallice sibbe gecyn</res>}
xpath3:	document(d_4)//dmg	{<dmg>w</dmg>, <dmg>de þa</dmg>}

Table 1: Results of XPath queries for question 5.

quite delicate. In case of question 5, the mediator must determine the **inclusion** relationship between elements returned by different XPath queries. Such determination, however, does not depend solely on the queries posed to the individual hierarchies. For example, question 2, "Find all words containing damaged characters," and question 3, "Find all words containing damaged characters ONLY," will be represented by the mediator with the same set of XPath queries: ***xpath1*** and ***xpath2***. However, the answers to these queries, are, in general, different. Therefore, during the merge operation the mediator has to deduce the exact semantics of the query from the incoming representation, and adjust its merge process according to it. All-in-all, this makes the mediator approach appear unnecessarily complicated, and still, depend on yet-to-be-named formalism for representing queries over distributed XML documents.

Method b) first merges the documents forming a distributed XML document D (following [9]). During the merge operation, some techniques, such as fragmentation, are used to resolve the overlapping conflicts. The resulting "master" XML document represents the same information content as D, and all individual component documents of D can be uniquely reproduced. However, there, typically, exist more than one possible way to convert D into a single XML document with such properties. The algorithm proposed in [9] attempts to create a compact "master" XML document, i.e., attempts to minimize the number of fragmentations to the best of its ability. However, from the structural point of view, it can lead to unexpected things. E.g., consider the following XML fragments from concurrent documents: d_1 =aaa<a>bbbbccc, d_2 =aaabbbbccc and d_3 =a<c>aab</c>bb<c>bcc</c>c. We observe that the content of <a> in d_1 is a subset of the content of in d_2, and therefore, we can think of <a> as the "descendent" of . However, because the markup in d_3 overlaps both <a> and , the following XML fragment d :

```
a<c>a<b l="1">a<a l="2">b</a></b></c>
<a l="2"><b l="1">bb</b></a>
<c><b l="1"><a l="2">b</a>c</b>c</c>c
```

is a possible output of the merging algorithm. Here, both <a> and tag have been fragmented into three fragments, but the middle fragment of <a> is *outside* the middle fragment of . What this means, is that XPath queries applied to the results of such mergers may yield unexpected results.

Based on the above observations, we envision a need for establishing an association between elements from different hierarchies that would not rely on either methods a) or b). We propose a flexible and efficient way to query and traverse distributed XML documents, that treats such documents as single objects, but operates on the underlying data model (GODDAG) rather than on its syntactic renderings. extend the standard XPath to allow us to query across concurrent hierarchies.

3 XPath for Concurrent Hierarchies

3.1 XPath

XPath is a language for addressing parts of an XML document [5]. Although it was initially designed to be used by XSLT and XPointer, XPath is intensively used as part of some XML query languages (XQuery), and can be used itself to query XML documents.

XPath uses a tree of nodes model to represent an XML document. There are seven types of nodes, the *root node* (a unique node in an XML document), *element, text, attribute, namespace, processing-instruction,* and *comment* nodes. The main syntactical construction of XPath is *expression*. An *expression* operates on a *context node* and manipulates *objects* of four kinds: node-set, boolean, string, and numeric.

The instrument for addressing sets of nodes in a document is the *location path* (a special kind of *expression*). A *location path* is composed of one or more *steps*, at each step a set of nodes is selected based on their relationship (specified in *step*) to each node in a current set of *context nodes*. The node set result of a *step* evaluation is the current set of *context nodes* for the next *step* in the *location path*. The first step in a *location path* is either relative to the current set of *context nodes* or absolute. For the later case, the current set of *context nodes* is the set of nodes containing only the *root node*. A *location path* syntax can be summarized as follows (the syntax is summarized, the comprehensive syntax is given in [5]):

```
locationPath := step1/step2/.../stepn
step := axis::node-test predicate*
predicate := [expression]
```

The main syntactical construction for a *step* evaluation is *axis*: for each node in the current *context node* set an *axis* is evaluated to a set of nodes according to

the respective *axis* definition. The set of nodes from *axis* evaluation is filtered by the *node-test* (basically a node type test or a name test for *element* nodes) and *expression* result in the context of each node of *axis* evaluation set (*axis* plays the selection role, *node-test* and *predicate* play the filtering role).

XPath uses 13 *axes* to address nodes in a document: *ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling,* and *self*. Axis names are rather verbose (in the context of tree model used by XPath) and their syntax and semantics are given in [5]. Formal semantics for XPath axes is also given in [28, 17].

For each XPath axis $\mathcal{A}$, we define the corresponding evaluation function for nodes in an XML document d,

$$\mathcal{A} : nodes(d) \rightarrow 2^{nodes(d)}$$

where $\mathcal{A}(x)$ evaluates axis $\mathcal{A}$ for the context node x.

XPath syntax also includes a core functions library for manipulating node-sets, strings, numerals, or boolean values. There are also two special functions, *position()* and *last()*, whose meanings make sense in the context of XML document order. Given a node x in a set of (distinct) nodes N, the former function returns an integer representing the position of x in N w.r.t. the document order and the latter function returns the size of the node-set N.

3.2 XPath axes extensions

Let $D = \langle d_1, \ldots, d_k \rangle$ be a distributed XML document over a concurrent XML hierarchy CMH. We define 11 new XPath axes, over the distributed document D, in the context of a node $x \in nodes(D)$: *xancestor, xdescendant, xancestor-or-self, xdescendant-or-self, xfollowing, xpreceding, following-overlapping, preceding-overlapping, overlapping, xancestor-or-overlapping*, and *xdescendant-or-overlapping*. As for XPath, for each extended axis, $\mathcal{X}$, we define the corresponding function

$$\mathcal{X} : nodes(D) \rightarrow 2^{nodes(D)}$$

where $\mathcal{X}(x)$ returns the set of nodes corresponding to axis $\mathcal{X}$ evaluated for the context of node $x \in nodes(D)$.

The new axes extend XPath in two ways. The first six axes from the list above are *extensions* of *ancestor, descendant, ancestor-or-self, descendant-or-self, following* and *preceding* axes onto the GODDAG model. The nodes obtained by evaluating each of these axes on a given node can be from any component of a distributed XML document. *Xancestor*/*xdescendant* axes are defined using superset/subset relation on the content of the nodes, represented via a set of *leaf nodes* in the GODDAG. To define *xfollowing* and *xpreceding* axes, we extend the document order onto the GODDAG. However, the document order over a GODDAG will no longer be total: overlapping markup will be incomparable. To capture this markup, we introduce two new base axes: *preceding-overlapping* and *following-overlapping*, and derive three more axes from them. The formal definitions for all new and extended axes are given below.

Definition 5 *The extensions of XPath axes in the context of a distributed document D and a node $x \in nodes(D)$ are defined as follows:*

1. $xancestor ::= ancestor(x) \cup \{y \in nodes(D - doc_D(x))|\ start(y) \leq start(x) \leq end(x) \leq end(y)\}$.
2. $xdescendant ::= descendant(x) \cup \{y \in nodes(D - doc_D(x))|\ start(x) \leq start(y) \leq end(y) \leq end(x)\}$.
3. $xancestor{-}or{-}self ::= xancestor(x) \cup \{x\}$.
4. $xdescendant{-}or{-}self ::= xdescendant(x) \cup \{x\}$.
5. $xfollowing ::= following(x) \cup \{y \in nodes(D - doc_D(x))|\ start(y) \geq end(x)\}$.
6. $xpreceding ::= preceding(x) \cup \{y \in nodes(D - doc_D(x))|\ end(y) \leq start(x)\}$.
7. $following{-}overlapping ::= \{y \in nodes(D)|\ start(x) < start(y) < end(x) < end(y)\}$.
8. $preceding{-}overlapping ::= \{y \in nodes(D)|\ start(y) < start(x) < end(y) < end(x)\}$.
9. $overlapping ::= following{-}overlapping(x) \cup preceeding{-}overlapping(x)\}$.
10. $xancestor{-}or{-}overlapping ::= xancestor(x) \cup overlapping(x)$.
11. $xdescendant{-}or{-}overlapping ::= xdescendant(x) \cup overlapping(x)$.

We give some examples of the extended axes, for the GODDAG in Figure 4 below.

(A) $xdescendant(line2) = \{t2, w4, w5, w6, t6, t7, t8, vline3, res3, t12, t13, dmg2, t17\}$

Node *line2* has one descendent in its document component, text node $t2$. The scope of $t2$ are leaf nodes $l7, \ldots, l11$. In every other component of D, we now search for nodes whose set of leaf nodes is a subset of $l7, \ldots, l11$. In $d2$, $t6$, $t7$ and $t8$ qualify, and so do their parents $w4$, $w5$ and $w6$. Going one level up, we discover that $vline3$ also is an *xdescendant* of $line2$, but $vline2$, the parent of $w4$ and $w5$ - is not, it's other child, $w3$ overlaps $line2$. Similar traversal of GODDAG parts corresponding to d_3 and d_4 yield remaining nodes in the answer set.

(B) $following{-}overlapping(w5) = \{dmg2, t17\}$

The content of $w5$ is leaf nodes $l9$ and $l10$. We are

looking for nodes in other components of D, whose content is $l10$ **and** $l11$. Such nodes, $t17$ and its parent $dmg2$,are found only in one component, d_4.

Remark. We note that Definition 5 allows a node x to be both an *xdescendant* and an *xancestor* of a node y: if $start(x) = start(y)$, $end(x) = end(y)$ and *they are in different documents.*

3.3 Extended XPath axes evaluation

Let $D = \langle d_1, \ldots, d_k \rangle$ be a distributed XML document and $\mathcal{X}$ – an extended XPath axis. We define extended axis restriction to each XML document in D as

$$\mathcal{X} : nodes(D) \times docs(D) \rightarrow 2^{nodes(d_j)}$$
$$\mathcal{X}(x, d_j) = \mathcal{X}(x) \cap nodes(d_j), \quad 1 \leq j \leq k.$$

The axes *xancestor, xdescendant, xancestor-or-self, xdescendant-or-self, xfollowing,* and *xpreceding* extend the semantics of their counterparts in regular Xpath. We say that such an extended axis $\mathcal{X}$ is dual to its respective regular XPath axis $\mathcal{A}$ and we denote this by $\mathcal{A} = dual(\mathcal{X})$. The following results follow directly from Definition 5 and show that the first six extended axes *specialize* to the semantics of their dual.

Theorem 1 *Let $\mathcal{X}$ denote one of the axes in Definition 5(1-6) and let $\mathcal{A} = dual(\mathcal{X})$. Then for a distributed XML document $D = \langle d_1, \ldots, d_k \rangle$ and $\forall x \in nodes(D)$,*

$$\mathcal{X}(x, doc_D(x)) = \mathcal{A}(x)$$

Corollary 1 *Let $\mathcal{X}$ denote one of the axes in Definition 5(1-6) and let $\mathcal{A} = dual(\mathcal{X})$. Then for a distributed XML document $D = \langle d \rangle$ and $\forall x \in nodes(D)$,*

$$\mathcal{X}(x) = \mathcal{A}(x)$$

An extended XPath axis is evaluated for each node in a *union of node-sets over the distributed document* and a *union of node-sets* corresponding to the evaluation is returned. Each node in the returned set of nodes constitutes the context node for the (possible) subsequent axis. In Table 2 we capture the semantics of the extended XPath axis evaluation, starting with a set of context nodes. However, a straightforward implementation of the semantics in Table 2 (for each node in the input set, evaluate the axis by visiting all nodes in the target document) would lead to a quadratic time complexity in the size of nodes in D.

$\mathcal{S} : Axis \rightarrow NodeSet \rightarrow NodeSet$
$\mathcal{S}[\![\mathcal{X}]\!](N_1 \cup \ldots \cup N_k) = \cup_{i=1}^{k} \mathcal{S}[\![\mathcal{X}]\!](N_i)$
$\mathcal{S}[\![\mathcal{X}]\!](N_i) = \cup_{j=1}^{k} \cup_{x \in N_i} \mathcal{X}(x, d_j)$

Table 2: Extended XPath axes evaluation

Before proceeding further, in order to precisely define a *step* evaluation, we need to give the semantics for a *predicate* evaluation. A *predicate* (and consequently an *expression*) is evaluated in a context of a node in a current node-set. An extended XPath axis evaluation holds a *union* of node-sets over the distributed document. The semantics of a predicate p evaluation for a node in the context of union of node-sets over a distributed document is summarized in Table 3.

$\mathcal{S} : Predicate \rightarrow (Node, \cup_1^k NodeSet) \rightarrow boolean$
$\mathcal{S}[\![P]\!](x, N_1 \cup \ldots \cup N_k) = \mathcal{S}[\![P]\!](x, N_j),\ x \in N_j$

Table 3: Extended XPath axes evaluation

Using the semantics in Table 3 it makes sense to use a function like *position()* which is strictly related to the document order (no total document order is defined over a distributed XML document).

Let $1 \leq i, j \leq k, i \neq j$, $x_1, x_2 \in nodes(d_i)$, $x_1 < x_2$. The following theorems establish important results about how the order of nodes in an axis evaluation (for a node $x \in d_i$ in a *target document* d_j) is related to the order of nodes in the input set of nodes.

Theorem 2 *Let $\mathcal{X}$ denote the* xancestor *axis.*

1. $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall z \in nodes(d_j)$, $start(z) > start(x_1) \vee start(x_1) \leq start(z) < end(z) < end(x_1)$ *then* $y_1 < z$.
2. *If* $start(x_1) < start(x_2)$, $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ *then* $y_1 < y_2$.
3. *If* $start(x_1) = start(x_2) < end(x_2) \leq end(x_1)$, *then* $\mathcal{X}(x_1, d_j) \subseteq \mathcal{X}(x_2, d_j)$.
4. *If* $start(x_1) = end(x_1)$, *then for* $\forall x \in nodes(d_i)$, $start(x_1) = start(x)$ *we have* $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x, d_j)$.

Proof. (1) We have $start(y_1) < start(z)$ or $start(y_1) \leq start(x_1) \leq start(z) < end(z) < end(x_1) \leq end(y_1)$, hence $y_1 < z$.
(2) We have $start(y_1) \leq start(x_1)$ and $start(x_1) \leq start(y_2) \leq start(x_1)$. If $start(x_1) < start(y_2)$ we are done: $start(y_1) < start(y_2)$ hence $y_1 < y_2$. If $start(x_1) = start(y_2)$ then we must have $end(y_2) < end(x_1) \leq end(y_1)$. It follows also that $y_1 < y_2$.
(3) Let $y_1 \in \mathcal{X}(x_1, d_j)$. We have $start(y_1) \leq start(x_1) = start(x_2) < end(x_2) \leq end(x_1) \leq end(y_1)$. Hence $y_1 \in \mathcal{X}(x_2, d_j)$, therefore $\mathcal{X}(x_1, d_j) \subseteq \mathcal{X}(x_2, d_j)$.
(4) Let $y \in \mathcal{X}(x, d_j)$. We have $start(y) \leq start(x) = start(x_1) = end(x_1) \leq end(x) \leq end(y)$. It follows that $y \in \mathcal{X}(x_1, d_j)$ hence $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x, d_j)$. □

Theorem 3 *Let $\mathcal{X}$ denote the* xdescendant *axis.*

1. $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall z \in nodes(d_j)$, $end(x_1) < start(z)$ *then* $y_1 < z$.
2. *If* $start(x_1) \leq start(x_2) \leq end(x_2) \leq end(x_1)$ *then* $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x_2, d_j)$.

3. *If* $end(x_1) = start(x_2)$ *then* $\mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j) = \{y \in nodes(d_j) : end(x_1) = start(y) = end(y) = start(x_2)\}$ *and* $\forall y_1 \in \mathcal{X}(x_1, d_j) - \mathcal{X}(x_2, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ *then* $y_1 < y_2$.

4. *If* $end(x_1) < start(x_2)$, $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j)$ *then* $y_1 < y_2$.

Proof. (1) We have $start(y_1) \leq end(x_1) < start(z)$. Hence $y_1 < z$.
(2) $\forall y \in \mathcal{X}(x_2, d_j)$ we have $start(x_1) \leq start(x_2) \leq start(y) \leq end(y) \leq end(x_2) \leq end(x_1)$. It follows that $y \in \mathcal{X}(x_1, d_j)$, thus $\mathcal{X}(x_1, d_j) \supseteq \mathcal{X}(x_2, d_j)$.
(3) $\forall y \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j)$, $y \neq NIL$ we have $start(x_1) \leq start(y) \leq end(x_1)$ and $start(x_2) \leq start(y) \leq end(x_2)$. It follows that $end(x_1) = start(y) = start(x_2)$. Similarly we get $end(x_1) = end(y) = start(x_2)$. Conversely, for $y \in nodes(d_j)$, $end(x_1) = start(y) = end(y) = start(x_2)$ it follows that $start(x_1) \leq start(y) \leq end(y) \leq end(x_1)$ and $start(x_2) \leq start(y) \leq end(y) \leq end(x_2)$, hence $y \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j)$.
Suppose that $\mathcal{X}(x_1, d_j) - \mathcal{X}(x_2, d_j) \neq \emptyset$, $\mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j) \neq \emptyset$. Then $start(y_1) \leq end(y_1) \leq end(x_1) \leq start(x_2) \leq start(y_2) \leq end(y_2)$. Moreover, $start(x_2) < end(y_2)$. Then *start* and *end tags* of y_2 must come after the *end tag* of y_1, so $y_1 < y_2$.
(4) We have $start(y_1) \leq end(x_1) < start(x_2) \leq start(y_2)$, hence $y_1 < y_2$. □

Theorem 4 *Let* $\mathcal{X}$ *denote the* xfollowing *axis and* $\mathcal{A}$ *denote the* following *axis. Let* $x \in nodes(d_i)$, $1 \leq i, j \leq k, i \neq j$, *and let* $y = parent(d_j, last{-}leaf(x))$. *Then* $\mathcal{X}(x, d_j) = \mathcal{A}(y)$.

Proof. Since $y = parent(d_j, last{-}leaf(x))$, then $end(x) \leq end(y)$.
Let $z \in \mathcal{A}(y)$. Then $end(y) \leq start(z)$. It follows that $end(x) \leq start(z)$, hence $z \in \mathcal{X}(x, d_j)$.
Conversely, let $z \in \mathcal{X}(x, d_j)$. Then $end(x) \leq start(z)$, so $start(y) < start(z)$. Since y is a parent of a *leaf* node, it must be a text node. So z is not a descendant of y. It follows that $z \in \mathcal{A}(y)$.
This proves that $\mathcal{X}(x, d_j) = \mathcal{A}(y)$. □

Theorem 5 *Let* $\mathcal{X}$ *denote the* xpreceding *axis and* $\mathcal{A}$ *denote the* preceding *axis. Let* $x \in nodes(d_i)$, $1 \leq i, j \leq k, i \neq j$, *and let* $y = parent(d_j, first{-}leaf(x))$. *Then* $\mathcal{X}(x, d_j) = \mathcal{A}(y)$.

Theorems 4 and 5 establish that *xfollowing* and *xpreceding* axes can be computed from *following* and *preceding* respectively, for appropriate context nodes in the target document.

For a node $x \in nodes(d_i)$ we define the *test nodes set* of x for the *following-overlapping* axis $\mathcal{X}$ in d_j as $\hat{\mathcal{X}}(x, d_j) = \mathcal{A}(parent(d_j, last{-}leaf(x_1))) \cap \{z \in d_j \mid start(x) < start(z) < end(x) \leq end(z)\}$.

Theorem 6 *Let* $\mathcal{X}$ *denote the* following-overlapping *axis and* $\mathcal{A}$ *denote the* ancestor-or-self *axis.*

1. $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.

2. *If* $start(x_1) \leq start(x_2) < end(x_2) = end(x_1)$ *then* $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.

3. *If* $end(x_1) \leq start(x_2)$ *then* $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j)$ *we have* $y_1 < y_2$ *and* $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

4. *If* $end(x_2) < end(x_1)$ *then* $\forall y_1 \in \mathcal{X}(x_1, d_j) \cap \mathcal{X}(x_2, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$ *then* $y_1 < y_2$.

Proof. (1) Let $y_1 \in \mathcal{X}(x_1, d_j)$, and $p = parent(d_j, last{-}leaf(x_1))$. We have that $start(x_1) < start(y_1) < end(x_1) < end(y_1)$ and $start(p) < end(x_1) \leq end(p)$. Hence $y_1 \notin following(p)$ and $p \notin following(y_1)$. Since p has no descendants (except for *leaf* nodes), it follows that p is a descendant of y. From Definition 5 it follows straight forward that $y_1 \in \{z \in d_j \mid start(x_1) < start(z) < end(x_1) \leq end(z)\}$. Hence $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.
(2) Let $y_2 \in \hat{\mathcal{X}}(x_2, d_j)$. We have $start(x_1) \leq start(x_2) < start(y_2) < end(x_2) = end(x_1) \leq end(y_2)$. It follows that $y_2 \in \hat{\mathcal{X}}(x_1, d_j)$, hence $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.
(3)We have $start(x_2) < start(y_1)$ and $start(y_1) < end(x_1) \leq start(x_2)$. Hence $y_1 < y_2$.
Let $z_1 \in \hat{\mathcal{X}}(x_1, d_j)$. Then $start(z_1) < end(x_1) \leq start(x_2)$. So $z_2 \notin \hat{\mathcal{X}}(x_1, d_j)$ and therefore $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.
(4) We have that $start(y_1) < end(x_2) < end(x_1) < end(y_2)$ and $start(y_2) < end(x_2) < end(y_2) \leq end(x_1)$. Hence $end(y_2) < end(y_1)$, therefore $y_1 < y_2$.
□

For a node $x \in nodes(d_i)$ we define the *test nodes set* of x for the *preceding-overlapping* axis $\mathcal{X}$ in d_j as $\hat{\mathcal{X}}(x, d_j) = ancestor{-}or{-}self(parent(d_j, first{-}leaf(x_1))) \cap \{z \in d_j \mid start(z) \leq start(x) < end(z) < end(x)\}$.

Theorem 7 *Let* $\mathcal{X}$ *denote the* preceding-overlapping *axis and* $\mathcal{A}$ *denote the* ancestor-or-self *axis.*

1. $\mathcal{X}(x_1, d_j) \subseteq \hat{\mathcal{X}}(x_1, d_j)$.

2. *If* $start(x_1) = start(x_2) < end(x_2) \leq end(x_1)$ *then* $\hat{\mathcal{X}}(x_1, d_j) \supseteq \hat{\mathcal{X}}(x_2, d_j)$.

3. *If* $end(x_1) \leq start(x_2)$ *then* $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

4. *If* $start(x_1) < start(x_2) < end(x_2) \leq end(x_1)$ *then* $\forall y_1 \in \mathcal{X}(x_1, d_j)$, $\forall y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j) : y_1 < y_2$. *Moreover, if* $\mathcal{X}(x_2, d_j) = \emptyset$ *then* $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.

Proof. (1) and (2) follow from using dual arguments as in the proof of Theorem 6.
(3) Let $y_1 \in \hat{\mathcal{X}}(x_1, d_j)$. We have $start(y_1) \leq start(x_1) < end(y_1) < end(x_1)$ and therefore $end(y_1) < end(x_1) \leq start(x_2)$. Hence $y_1 \notin \hat{\mathcal{X}}(x_2, d_j)$. It follows that $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$.
(4) Let $y_1 \in \mathcal{X}(x_1, d_j)$, $y_2 \in \mathcal{X}(x_2, d_j) - \mathcal{X}(x_1, d_j)$. We have that $start(y_1) < start(x_1)$ and $start(x_1) \leq start(y_2) < start(x_2)$. Hence $y_1 < y_2$.
Let $z \in \hat{\mathcal{X}}(x_2, d_j)$. If $\mathcal{X}(x_2, d_j) = \emptyset$ then $start(z) = start(x_2) < end(z) < end(x_2)$. Hence $start(x_1) < start(z)$ therefore $z \notin \hat{\mathcal{X}}(x_1, d_j)$. It follows that $\hat{\mathcal{X}}(x_1, d_j) \cap \hat{\mathcal{X}}(x_2, d_j) = \emptyset$. □

4 Algorithms for Evaluation of XPath Extended Axes over Concurrent XML Markup

Polynomial time evaluation algorithms for XPath queries, using DOM representation of an XML Document, are given in [16, 17]. An algorithm that evaluates XPath axes in linear time (in the size of nodes in the input XML document) is also given in [17].

In this section we present linear time complexity algorithms for evaluation of *xancestor*, *xdescendant*, *following-overlapping*, and *preceding-overlapping* for a set of nodes in a distributed XML document. As stated in theorems 4 and 5, *xfollowing* and *xpreceding* can be computed in a straightforward manner using the algorithms for efficient evaluations of the *following* and *preceding* axes ([17] gives linear time algorithms for this). Finally, *xancestor-or-self*, *xdescendant-or-self*, *overlapping*, *xancestor-or-overlapping*, and *xdescendant-or-overlapping* axes are derivatives of the other six, and their computation a simple set union operation.

Figure 5 shows the pseudocode of the four algorithms for evaluation of *xancestor*, *xdescendant*, *following-overlapping*, and *preceding-overlapping* axes, and a wrapper algorithm evaluating the axes on a node set. Each of the algorithms 1, 2, 3, and 4 takes as the input a set of nodes $N_i \subseteq nodes(d_i)$, in increasing order in d_i, and a *target document* d_j where the output set of nodes N'_j is to be evaluated. In each algorithm pseudocode we use keywords like *isxancestor*, *isxdescendant* etc., as shortcuts of the tests in Definition 5 for the respective axis (*xancestor*, *xdescendant*, etc.).

Algorithm 1 evaluates the *xancestor* axis. The algorithm selects first only the nodes that start at distinct positions (lines 5-9), as nodes starting at the same position have overlapping *xancestors* (Theorem 2(3 and 4)). Then the *xancestor* nodes are added to the output set of nodes as each node in *target document* d_j is visited. Each node in N_i and each node in d_j is visited once. The correctness of this algorithm follows from Theorem 2.

Algorithm 2 evaluates the set of *xdescendant* nodes in d_j for each node in the input set N_i. The algorithm checks for each node $y \in nodes(d_j)$ whether or not y is an *xdescendant* of a node in N_i. Each node in N_i, as well as each node in $nodes(d_j)$ is visited only once. The correctness of the algorithm follows from Theorem 3. Nodes in N_i and nodes in d_j are visited in their document order, based on the order preserving properties in Theorem 3(3,4). Special treatment for nodes without text node descendants (*start* and *end* tags are at the same position) is implemented in lines 5-7.

Algorithm 3 computes the *following-overlapping* set of nodes for an input set $N_i \subseteq d_i$ and a *target document* d_j. For each node x in the input set, all *test nodes* of x in d_j are examined. The correctness of the algorithm follows from Theorem 6. The algorithm uses a stack mechanism (lines 6-18) to ensure that for any node $x \in N_i$, the descendants of x that have the same *end tag* position as x are skipped (cf. Theorem 6(2), these nodes have the same *test nodes set* as x). As it follows from Theorem 6(4) it might be the case that some of the nodes in d_j visited as *test nodes* for some node $x \in N_i$ are visited again as *test nodes* for some of the children of x. This happens if x has no *following-overlapping* nodes but some child does. However, this situation does not propagate down to x's other descendants (since the respective child node of x has *following-overlapping nodes*). Moreover, different children of x visit different nodes in the *test nodes set* of x. Consequently, no node in d_j is visited more than two times.

Algorithm 4 evaluates *preceding-overlapping* axis for a set of nodes $N_i \in nodes(d_i)$ and a *target document* d_j, $i \neq j$. The algorithm finds the *preceding-overlapping* nodes for each node x in the input by examining the *test nodes set* for x in d_j (correctness follows from Theorem 7(1). Similarly to Algorithm 3, there are cases (Theorem 7(4)) when some of the nodes in d_j are visited more than once. A argument mirroring the one for Algorithm 3, can easily lead to the conclusion that no node in d_j is visited more than two times.

Theorem 8 *Algorithms 1, 2, 3, and 4 evaluate extended XPath axes for a set of nodes $N_i \subseteq nodes(d_i)$ and a* target document *d_j in $O(|N_i| + |nodes(d_j)|)$ time.*

Proof. The proof follows directly from the fact that each node in N_i is visited once and each node in d_j is visited at most twice. □

Theorem 9 *Algorithm 5 computes $\mathcal{X}(N)$, $N \subseteq nodes(D)$, in $O(k|nodes(D)|)$ time.*

Proof. We have $N'_j = \cup_{i=1, i\neq j}^{k} \mathcal{X}(N_i, d_j) \cup \mathcal{A}(N_j)$. This adds up to $\sum_{i=1}^{k} O(|N_i| + |nodes(d_j)|) = O(|nodes(D)| + k|nodes(d_j)|)$ time. Hence the overall evaluation of $N' = N'_1 \cup \ldots \cup N'_k$ takes

Algorithm 1: `xancestor` evaluation

XANCESTOR(N_i, d_j)
(1) $N'_j = \emptyset$
(2) $x \leftarrow first\ node\ in\ N_i$
(3) $y \leftarrow root(d_j)$
(4) **while** $y \neq NIL \wedge x \neq NIL$
(5) **while** next node in N_i starts at $start(x)$
(6) **if** $start(x) \neq end(x)$
(7) $x \leftarrow next\ node\ in\ N_i$
(8) **else**
(9) $next\ node\ in\ N_i$
(10) **if** y isxancestor of x
(11) $append\ y\ to\ N'_j$
(12) $y \leftarrow next\ node\ in\ d_j$
(13) **else if** $start(x) < start(y) \vee start(x) < end(y) < end(x)$
(14) $x \leftarrow next\ node\ in\ N_i$
(15) **else**
(16) $y \leftarrow next\ node\ in\ d_j$
(17) **return** N'_j

Algorithm 2: `xdescendant` evaluation

XDESCENDANT(N_i, d_j)
(1) $N'_j = \emptyset$
(2) $x \leftarrow first\ node\ in\ N_i$
(3) $y \leftarrow root(d_j)$
(4) **while** $y \neq NIL \wedge x \neq NIL$
(5) **if** $start(x) = end(x)$ and the next node in N_i starts at $start(x)$
(6) $x \leftarrow next\ node\ in\ N_i$
(7) **continue**
(8) **if** y isxdescendant of x
(9) $append\ y\ to\ N'_j$
(10) $y \leftarrow next\ node\ in\ d_j$
(11) **else if** $end(x) < start(y)$
(12) $x \leftarrow next\ node\ in\ N_i$
(13) **else**
(14) $y \leftarrow next\ node\ in\ d_j$
(15) **return** N'_j

Algorithm 3: `following-overlapping` evaluation

$following{-}overlapping(N_i, d_j)$
(1) $N'_j = \emptyset$
(2) $stack\ S = empty$
(3) $index \leftarrow 0$
(4) $S.push(index)$
(5) **foreach** $x \in N_i,\ start(x) < end(x)$
(6) **if** $end(x) = index$
(7) **continue**
(8) **else if** $end(x) > index$
(9) $index \leftarrow end(x)$
(10) $i \leftarrow S.peek()$
(11) **if** $index > i$
(12) $S.pop()$
(13) **else if** $index = i$
(14) $S.pop()$
(15) **continue**
(16) **else**
(17) $S.push(index)$
(18) $index = end(x)$
(19) **foreach** $y \in \hat{\mathcal{X}}(x, d_j)$
(20) **if** y isfollowing-overlapping x
(21) **if** $y \in N'_j$
(22) **break**
(23) append y to N'_j
(24) **return** N'_j

Algorithm 4: `preceding-overlapping` evaluation

$following{-}overlapping(N_i, d_j)$
(1) $N'_j = \emptyset$
(2) $current{-}index \leftarrow 0$
(3) **foreach** $x \in N_i,\ start(x) < end(x)$
(4) **foreach** $y \in \hat{\mathcal{X}}(x, d_j)$
(5) **if** $start(y) = current{-}index$
(6) **continue**
(7) **if** y ispreceding-overlapping x
(8) **if** $y \in N'_j$
(9) **break**
(10) append y to N'_j
(11) $current{-}index \leftarrow start(y)$
(12) **return** N'_j

Algorithm 5: Extended XPath axis evaluation
Input: $N = N_1 \cup \ldots \cup N_k$
Output: $N' = N'_1 \cup \ldots \cup N'_k$
$\mathcal{X}evaluation(N)$
(1) **for** $j = 1$ **to** k
(2) $N'_j = \emptyset$
(3) **for** $i = 1$ **to** k, $i \neq j$
(4) $N'_j \leftarrow N'_j \cup \mathcal{X}(N_i, d_j)$
(5) $N'_j \leftarrow N'_j \cup \mathcal{A}(N_i)$
(6) **return** $N'_1 \cup \ldots \cup N'_k$

Figure 5: Algorithms for evaluation of Extended XPath axes.

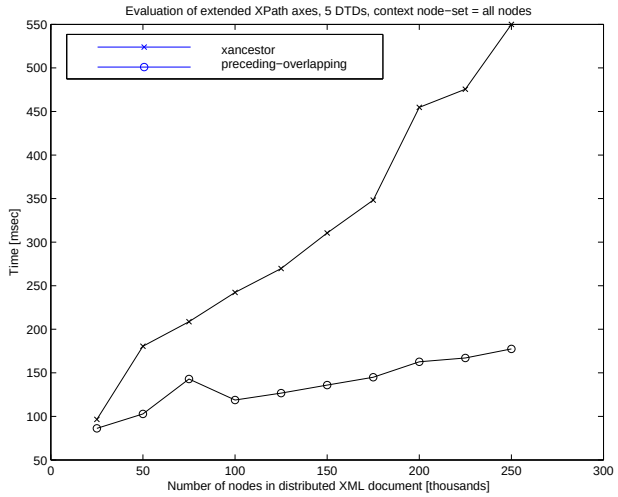


Figure 6: Evaluating *xancestor* and *preceding-overlapping*.

$\sum_{j=1}^{k} O(|nodes(D)|+k|nodes(d_j)|) = O(k|nodes(D)|)$ time.

We need to conclude this section by specifying how the ordered set of nodes $N'_1, \ldots, N'_k$ can be obtained as the output of each *axis* evaluation within the time complexity bounds established by Theorem 9. Note that algorithms 1-4 use ordered node-sets as input but they not necessarily produce output node-sets in document order (per each document). Since the output node-sets of a *step* evaluation are the input of the subsequent *step* we need to make sure that Algorithm 5 (Figure 5) produces ordered node-sets. This property can be easily achieved by regenerating the output node-sets $N''_1, \ldots, N''_k$ from the output $N'_1, \ldots, N'_k$ of Algorithm 5 as follows: for each document d_j, $1 \leq j \leq k$, visit all nodes in document order and output in N''_j only nodes in N'_j. This operation takes no more than $O(|nodes(D)|)$ time, hence the complexity bound in Theorem 9 still holds.

5 Experimental results

We have implemented and tested the algorithms described in Section 4. Figure 6 shows the results of testing two of the extended XPath axes: *xancestor* and *preceding-overlapping*. The algorithms were implemented in Java and run on a Dell GX240 PC with 1.4Ghz Pentium 4 processor and 256 Mb main memory. The tests were run on randomly generated distributed XML documents with 5 components over 100,000 character content string[3]. As the independent variable we chose the number of nodes in the GODDAG of the distributed XML document. For each tested value, we have generated 10 distributed XML documents and evaluated the extended XPath axes on a context node set that included *all* nodes in the GODDAG. The average times obtained in the test are reported in graph in Figure 6. The results show that generally, the time it takes to evaluate one axis is linear in the size of the GODDAG.

[3]In our tests we have used the same DTD repeated five times, with tags renamed in each copy. The DTD contained 10 XML elements. The XML files generated for the tests ranged from 135Kb to 1MB.

6 Related work

Related work of representing concurrent XML markup was already discussed in some detail in Section 1.

Querying a distributed XML document is similar in some aspects with data integration and querying XML over multiple data sources. A data integration system provides common interfaces for data sources collections [22]. Queries in a data integration system are answered by combining data from underlaying systems. Usually, a mediated schema is developed by the system designer and schema mappings (*source descriptions*) are given from each data source schema to mediated schema [11]. There are two general schemas for querying multiple data sources: the *global-as-view* (GAV) and *local-as-view* (LAV). The GAV approach defines a global schema from given local schemas, while LAV defines local schemas as views over a given global schema [23].

In our approach, a complex document-centric XML document is viewed as a (virtual) collection of (lower schema complexity) documents, the *distributed XML document*. Each *step* of a *location path* using a extended XPath axis is locally evaluated (for each component document of the distributed document) then the results are merged. This approach benefits on GAV superiority on query translation to local data sources: this is straight forward from extended axes semantics (there is no need of a mediated layer). However, as opposed to GAV, there is no need of a global schema in order to query the distributed XML document. Another important difference is that we don't necessarily deal with multiple data sources: a *distributed XML document* is rather a virtual than a physical representation of a collection of XML documents. A single, complex, document-centric XML document which uses *fragmentation* or *milestone elements* to resolve markup conflicts can be represented as a distributed XML document then queried using extended XPath.

The data structure we use to speed up the extended XPath axes evaluation uses a partial *region encoding* of an XML document [14, 18, 4]. *Region encoding* is a technique frequently used in indexing XML documents [4, 18] to speed up *structural joins*, considered to be core operations in XML queries optimization [14, 25, 2, 4]. Generally, a document *region encoding* is a hyperplane of nodes, each node being described by three parameters (based on the tree representation of the document): pre-order traversal index, post-order

traversal index, and the depth of the node. Based on these node parameters, XPath axes can be efficiently evaluated using R-tree [4, 18] or B-tree [4] indexing.

7 Conclusions

This paper proposes extensions for XPath axes to be used in the context of concurrent markup in complex document-centric XML documents. The solution we propose is applicable, but not limited, to a novel *distributed XML document* representation. A *distributed XML document* is a virtual rather than a physical representation of a collection of XML documents. Single document-centric XML documents using complex general schema and *fragmentation* or *milestone elements* to resolve markup conflicts can be parsed into a distributed XML document according to smaller, simpler sub-schemas of the general schema.

Our in-memory algorithms for evaluating the new axes have linear-time combined complexity (in the size of documents and the number of documents), hence as efficient as other known algorithms for XPath axes evaluation [15, 17].

In the future we intend to work on improving running time lower bounds for the extended XPath axes evaluation algorithms and creating a database index structure for efficient querying distributed XML documents stored on disk.

References

[1] British Library MS Cotton Otho A. vi, fol. 36v.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *In Proc. of ICDE*, 2002.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler(Eds.). Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, Oct 2000. W3C, REC-xml-20001006.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, and V. J. Tsotras. Efficient structural joins on indexed XML documents. In *In Int. Conf. on Very Large Data Bases (VLDB)*, pages 263–274, 2002.

[5] World Wide Web Consortium. XML Path Language (XPath) (Version 1.0). http://www.w3.org/TR/xpath, Nov 1999. W3C, REC-xpath-19991116.

[6] World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, Nov 1999. W3C Recommendation.

[7] World Wide Web Consortium. XML Pointer Language (XPointer) Version 1.0. http://www.w3.org/TR/WD-xptr, Jan 2001. Working Draft.

[8] World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, Nov 2003. Working Draft.

[9] A. Dekhtyar and I. E. Iacob. A Framework for Management of Concurrent XML Markup. In *Proc., XSDM*, 2003.

[10] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 431–442, 1999.

[11] AnHai Doan, Pedro Domingos, and Alon Y. Levy. Learning source description for data integration. In *WebDB (Informal Proceedings)*, pages 81–86, 2000.

[12] P. Durusau and M.B. O'Donnel. Declaring Trees: The Future of the Evolution of Markup? In *Proc. Conference on Extreme Markup Languages*, 2002.

[13] P. Durusau and M. B. O'Donnell. Concurrent Markup for XML Documents. In *Proc. XML Europe*, May 2002.

[14] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[15] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, San Diego, CA.*, pages 179–190, June 2003.

[16] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space eficiency. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03), Bangalore, India.*, pages 379–390, Mar 2003.

[17] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.

[18] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM Press, 2002.

[19] C. Huitfeldt and C. M. Sperberg-McQueen. TexMECS: An experimental markup meta-language for complex documents. http://www.hit.uib.no/claus/mlcd/papers/texmecs.html, February 2001.

[20] K. Kiernan. Digital Facsimiles in Editing: Some Guidelines for Editors of Image-based Scholarly Editions. *Electronic Textual Editing*, 2004. forthcoming.

[21] K. Kiernan, J. Jaromczyk, A. Dekhtyar, D. Porter, K. Hawley, S. Bodapati, and I. Iacob. The ARCHway project: Architecture for research in computing for humanities through research, teaching, and learning. *Literary and Linguistic Computing*, 2004. forthcoming.

[22] Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[23] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries over heterogeneous data sources. In *Proc. of the Int'l. Conf. on Very Large Databases (VLDB) , Roma, Italy*, pages 241–250, 2001.

[24] A. Renear, E. Mylonas, and D. Durand. Refining our notion of what text really is: The problem of overlapping hierarchies. *Research in Humanities Computing*, 1993. N. Ide and S. Hockey, (Eds.).

[25] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[26] C. M. Sperberg-McQueen and L. Burnard(Eds.). Guidelines for Text Encoding and Interchange (P4). http://www.tei-c.org/P4X/index.html, 2001. The TEI Consortium.

[27] C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies, Sept. 2000. Early draft presented at the ACH-ALLC Conference in Charlottesville, June 1999.

[28] P. Wadler. Two semantics for XPath, 1999. http://www.cs.bell-labs.com/who/wadler/topics/xml.html.

[29] A. Witt. Meaning and interpretation of concurrent markup. In *Proc., Joint Conference of the ALLC and ACH*, pages 145–147, 2002.