Towards a Query Language for Multihierarchical XML: Revisiting XPath

Ionut E. Iacob[*]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
eiaco0@cs.uky.edu

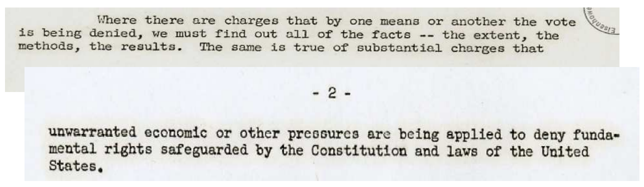
Alex Dekhtyar[†]
Department of Computer Science
University of Kentucky
Lexington, KY, USA
dekhtyar@cs.uky.edu

ABSTRACT

In recent years it has been argued that when XML encodings become complex, DOM trees are no longer adequate for query processing. Alternative representations of XML documents, such as multi-colored trees [7] have been proposed as a replacement for DOM trees for complex markup. In this paper we consider the use of Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs) for the purpose of storing and querying complex document-centric XML. GODDAGs are designed to store multihierarchical XML markup over the shared PCDATA content. They support representation of overlapping markup, which otherwise cannot be represented easily in DOM. We describe how the semantics of XPath axes can be modified to define path expressions over GODDAG, and enhance it with the facilities to traverse and query overlapping markup. We provide efficient algorithms for axis evaluation over GODDAG and describe the implementation of the query processor based on our definitions and algorithms.

1. INTRODUCTION

XML has become a popular approach to storage and transfer of diverse data because of its simplicity and transparency, as well as because of wide availability of (free) tools for working with it. Availability of open-source XML-related standards allows software developers build XML-enabled applications in a straightforward manner: XML files are parsed using a combination of SAX and DOM parsers, constructed memory-resident DOM trees are accessed from applications via DOM API calls. More complex XML management tasks involve the use of XPath and/or XQuery expressions for querying the content of DOM trees and XSLT for converting the content/structure of the tree, usually, for the purpose of visualizing the data. For simple XML data, XPath/XQuery over DOM Trees provide efficient and convenient way for querying.



```
          Where there are charges that by one means or another the vote
is being denied, we must find out all of the facts -- the extent, the
methods, the results.  The same is true of substantial charges that

                                  - 2 -

unwarranted economic or other pressures are being applied to deny funda-
mental rights safeguarded by the Constitution and laws of the United
States.
```

Figure 1: A fragment of a letter detailing the proposed Civil Rights Program to the members of President Eisenhower's Cabinet.

However, straightforward approaches to organizing XML for querying might yield unsatisfactory solutions when complex markup is considered. In [7] Jagadish et al. observed that querying XML data in the presence of several hierarchies for encoding features of the same objects can be done more efficiently if alternative data structures are used in place of a set of independent DOM trees, one for each of the hierarchies. Jagadish et al. proposed a data structure called *multi-colored tree* (MCT) for storing such markup and discussed efficient query evaluation strategies.

The approach of [7] was designed with data-centric XML in mind. The multicolored tree structure is built on top of individual XML nodes. This allows hierarchies of different "colors" to share content of some of the nodes. When *document-centric* XML is considered, however, there is an additional dimension, not captured by MCTs: the *sharing* of information in the hierarchies occurs at the level of content, rather than XML elements. Indeed, typically, document-centric XML documents are built by starting with a text and introducing various markup *on top* of it. When more than one hierarchy is used to encode features of a text, often the scopes of different markup elements *overlap*. This is illustrated on the following example.

EXAMPLE 1. *Consider a fragment of [8], shown in Figure 1. We describe two markup hierarchies for this document. First hierarchy describes, using elements* `<p>` *(paragraph),* `<sentence>` *and* `<w>` *(word), the structure of the text of the document. Our second hierarchy uses elements* `<page>` *and* `<line>` *to describe the physical layout of the text. The (somewhat simplified) corresponding XML encodings of this fragment are shown in Figures 2.(a) and 2.(b). Examination of the scopes of the XML elements in these figures reveals numerous overlapping conflicts. In particular, we mention the conflict between the scope of the* `<page no="1">` *element and the content of both* `<p>` *and* `<sentence no="14">` *elements. Similarly, the* `<w>` *element around the word* `"fundamental"` *over-*

---

[*]Work supported, in part, by the NEH grant RZ-20887-02.

[†]Work supported, in part, by the NSF grant ITR-0219924 and the NSF grant ITR-0325063.

Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland, USA.
.

laps both `<line no="1">` *and* `<line no="2">` *elements of* `<page no="2">`*. Overall, even this simple fragment, described using just two hierarchies contains six pairs of elements with overlapping content.*

Overlap in content of elements means that the markup presented in Figure 2 cannot be stored in a single XML document/DOM tree in a straightforward manner. As they lack facilities to store overlapping markup, it also cannot be stored in a single MCT. At the same time, storing each hierarchy in a separate DOM tree is inefficient from the perspective of query processing. For example, a user query

> Find all sentences completely or partially located on page 1, which contain the word "charges"

requires navigation through both text structure and physical location markup. Similar to the cases considered in [7], executing such a query as a join is inefficient. To make matters worse, the full answer to this query must include sentence number 14, only partially located on page 1. This means that the abovementioned query is not expressible in XPath (or XQuery, for that matter) over the set of the two encodings in Figure 2[1].

In [9], Sperberg-McQueen and Huitfeldt have introduced Generalized Ordered-Descendant Directed Acyclic Graphs (GODDAGs), a data structure for storing concurrent/ multihierarchical markup. A GODDAG combines DOM trees of individual XML hierarchies together by "tying" them at the top, root level, and at the bottom, content level. In [9], Sperberg-McQueen cite the need for appropriate mechanisms for building GODDAGs and querying data stored in them. The former problem had been addressed in [6]. In this paper, we adopt GODDAG (formally described in Section 2) as the data structure for storing concurrent markup. We then proceed to: (i) define the semantics of XPath axes over multiple hierarchies in GODDAG structures (Section 3); (ii) enhance XPath syntax and semantics with constructs for capturing overlapping markup (Section 3); (iii) develop and implement algorithms for axis evaluation over GODDAG[2] and conduct a preliminary study of the efficiency of enhanced XPath over GODDAG as the means of querying multihierarchical, overlapping markup (Section 4).

This paper describes the first steps toward a query language for document-centric XML data with overlapping hierarchies. We give an extension of XPath, as a navigational language through a data structure that we consider appropriate for representing multihierarchical markup. The next step, currently under development, would be to use this XPath extension in an XQuery language extension for querying multihierarchical XML documents.

2. DATA STRUCTURE FOR OVERLAPPING HIERARCHIES

We identify three basic principles for choosing a data structure for overlapping hierarchies: (i) we want to preserve individual hierarchies inside the complete document representation, (ii) we want to easily navigate from one structure to another, and (iii) we want to capture relationships between elements in different hierarchies.

Is is a fact that complex queries are likely to be expensive ([3, 2]). In [7] it is pointed out that, even for complex hierarchies, a tree-like structure is desirable due its relative navigation simplicity.

We start by introducing *concurrent markup hierarchies* and *distributed XML documents*. A concurrent markup hierarchy (CMH) is a collection of schema definitions (DTDs, XSchemas, etc...) that share a single (root) element name, and only it[3]. Individual schemas are called *hierarchies*. Given a CMH $C = \langle T_1, \ldots, T_k \rangle$, a distributed XML document (DXD) D over C is a collection of XML documents $(d_1, \ldots, d_k)$, one for each hierarchy of C, such that all documents **have the same PCDATA content**[4] Individual documents d_i are called *components* of D. They are not expected to be valid w.r.t. their schema T_i, but must contain markup only from T_i. This separation of markup in a DXD addresses principle (i) above: each document preserves the structure of the specific encoding. Two XML documents in Figure 2 show us an example of a DXD with two document components: d_1 on top (corresponding to a "text" hierarchy), and d_2 at the bottom (corresponding to a "physical layout" hierarchy). As clear from this example, DXDs can incorporate within them overlapping markup.

Representing DXD components as individual independent DOM trees is inconvenient, as illustrated in [7]. Instead, we use a structure called *General Ordered-Descendant Directed Acyclic Graph (GODDAG)*, originally introduced by Sperberg-McQueen and Huitfeldt in [9] precisely for the purpose of storing concurrent markup. Informally, a GODDAG for a distributed XML document D 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 (PCDATA). Because of possible overlap in the scopes of XML elements (text nodes) from different component documents, the underlying content of the document is stored *not* in text nodes, but in a special *new type* of node called *leaf node*. 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 from* **any** of the components of the distributed XML document. While each component of D will has its own text nodes in a GODDAG, the leaf nodes will be shared among all of them. As a consequence, **leaf nodes have multiple parents**: one in each component of D.

The GODDAG for the DXD in Figure 2 is illustrated in Figure 3. In the figure, nodes in the "text" hierarchy are on the top part, whereas nodes for the "physical layout" hierarchy are at the bottom. Leaf nodes are represented in the middle as rectangles corresponding to the PCDATA they cover. Element nodes are explicitly drawn with names and attribute values. Text nodes are symbolized by T in a circle. To easily identify the nodes, we put a unique label next to each node. Note here, for example, that the word "fundamental" is broken into two leaf nodes: L12:"funda" and L13:"mental". This allows us to represent the content of the appropriate `<w>` element (116) in the first hierarchy as {L12, L13}, while including L12 and L13 in the scope of two different `<line>` elements (29 and 211 respectively).

To define GODDAG formally, we need to introduce some notation. For an XML document d we let $root(d)$ denote the root element of d and $nodes(d)$ – the set of all nodes in DOM of d. For a node $x \in nodes(d)$ we let $string(x)$ be the PCDATA content of x (as defined in XPath [1]). We also set $start, end : nodes(d) \to \mathbb{N}$ to return the offset positions in $string(root(d))$ of *start tag* and *end tag* respectively for a node $x \in nodes(d)$. If x is a text node, then $start(x)$, $end(x)$ denote the start offset and end offset respectively. For a distributed document D we let $leaves(D)$ represent

[1]We note that it is possible to represent the desired query in XQuery by modifying the representation in Figure 2 in a number of ways, e.g., with ID/IDREF attributes, or with `<leaf>` elements representing GODDAG leafs described elsewhere in the paper.

[2]Due to the lack of space we will not present the algorithms here. The details can be found in [5].

[3]Namespaces can be used to distinguish elements from different hierarchies with the same name, but this fact is not important for the scope of this paper.

[4]The order of characters in the PCDATA content of all documents must be the same.

```
<doc id="CP56483">
...
<p>
<sentence no ="13"> <w>Where</w> <w>there</w> are
  <w>charges</w> that by one means of another the vote
  is being denied, we must find out all of the
  facts -- the extent, the methods, the results.
</sentence>
<sentence no="14">The same is true of substantial
  <w>charges</w> that unwarranted economic of other
  pressures are being applied to deny
  <w>fundamental</w> <w>rights</w> <w>safeguarded</w>
  by the Constitution and laws of the United States.
</sentence>
</p>
...</doc>
```

(a)

```
<doc id="CP56483">
...
<page no="1">
<line no="31"Where there are charges that by
      one means of another the vote</line>
<line no="32">is being denied, we must find out
      all of the facts -- the extent, the</line>
<line no="33">methods, the results. The same is true of
      substantial chargers that</line>
</page>
<page no="2">
<line no="1">unwarranted economic of other pressures are
      being applied to deny funda</line>
<line no="2">mental rights safeguarded
      by the Constitution and laws of the United</line>
<line no="3"> States.
...
</page> ...</doc>
```

(b)

Figure 2: Encoding of the fragment from Figure 1: (a) text structure, (b) physical location.

the set of all *leaf nodes* in D and we extend the domain of functions $string$, $start$, and end over the $leaves(D)$ set. For *leaf nodes* these functions are defined in the same way as for *text nodes*.

DEFINITION 1. *Let $D = (d_1, \ldots, d_k)$ 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:*
- $N = \cup_{i=1}^{k} nodes(d_i) \cup leaves(D)$
- $E = \cup_{i=1}^{k}\{(x,y) | x, y \in nodes(d_i) \wedge$ *x is the parent of y*$\} \cup$ $\cup_{i=1}^{k}\{(x,y) | x \in nodes(d_i)$ *is a text node*, $y \in leaves(D) \wedge$ $start(x) \le start(y) < end(y) \le end(x)\}$

The GODDAG data structure solves nicely the problem of navigation between CMH structures (principle (ii)): all hierarchies are connected via the common root node and common leaf nodes. The data structure also captures relationships among features in different structures. For instance, in the GODDAG in Figure 3, we can find all sentences partially or totally located on page 1: from `<page no="1">` (node 21) we navigate down and find all leaf nodes it contains (leaves L1 to L10); then we navigate up in the other hierarchy and find all `<sentence>` ancestors for these leaf nodes (nodes 13 and 14). In fact, as we show below, the semantics of all standard relationships between elements from different hierarchies (ancestor, descendant, overlapping, following, preceding) *can be expressed in terms of relationship between the corresponding leaf nodes*.

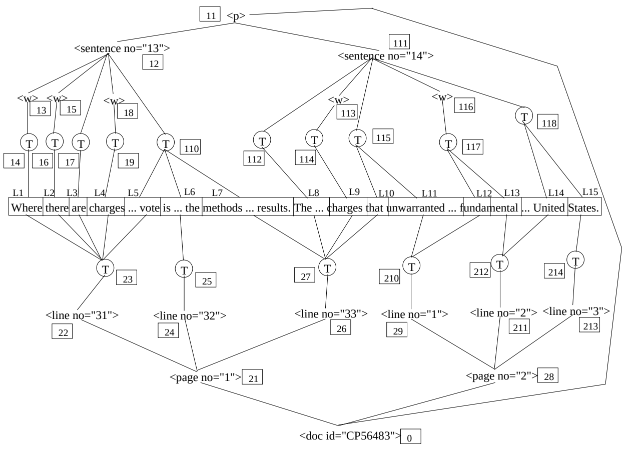


Figure 3: A GODDAG for the distributed document DXD in Figure 2

3. QUERYING DISTRIBUTED XML DOCUMENTS

XPath is a language for addressing parts of an XML document. It is intensively used as part of some XML query languages (XQuery), and can be used itself to query XML documents. In fact, in XQuery queries, XPath expressions are responsible for traversing the underlying XML document model (DOM tree) to discover requested XML nodes.

We argue in [5] that even simple queries are hard to express in XPath over representations of concurrent hierarchies that involve markup fragmentation or empty elements to overcome markup conflicts. In this section we show that when distributed XML documents are represented in GODDAG structures, we can express such queries as path expressions in straightforward ways. In addition, we show that individual components of path expressions (we concentrate on axes) have natural semantics over GODDAG, a semantics that specializes to XPath over DOM semantics when single-component documents are considered. We start by discussing how individual XPath axes can be defined in GODDAGs then we introduce formal definitions of XPath components over GODDAG.

3.1 Path Expressions Over GODDAG

Recall that 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* composed of one or more *steps*. Each step consists of an *axis*, a *nodetest* and zero or more *predicates*. An axis determines the direction of traversal from the current (context) node, while nodetests and predicates filter nodes that do not match them. A *location path* syntax can be summarized as follows (comprehensive syntax is given in [1]):

$\texttt{locationPath := step}_1\texttt{/step}_2\texttt{/.../step}_n$
`step := axis::node-test predicate*`
`predicate := [expression]`

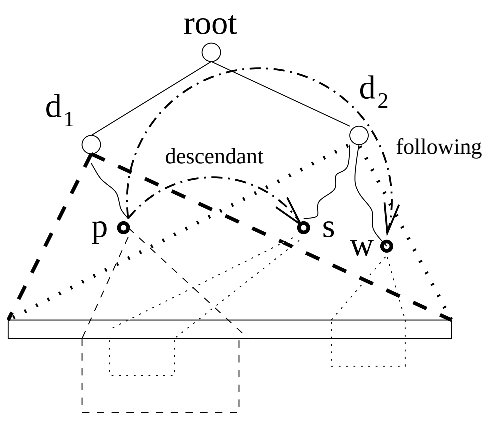


Figure 4: Descendant and following axes in GODDAG.

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 (evaluated to *true* or *false*) 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.*

We illustrate the problem of definition of XPath axes over a GODDAG in the following examples. In the context of our example from Figures 1 and 2, consider the query "Find all sentences completely inside page 1". If `<page>` and `<sentence>` markup were in the same hierarchy, we would have expressed this query using the following XPath expression:

`//page[attribute::no="1"]/descendant::sentence`

But in our GODDAG (Figure 3), they are not. Yet, the representation mechanism should not affect our understanding of the relationship between pages and sentences. By definition of the *descendant* axis, a `<sentence>` node is a descendent of a `<page>` node if it is located in the `<page>` node's subtree. However, we can describe a descendant relationship in a different way:

> a node x is a descendant of node y iff the content range of x is *completely included* in the content range of y.

When considered over DOM trees, these two definitions are (almost) equivalent. The key difference between them is that while the former definition is DOM-specific, the latter is not. In Figure 4 we show the latter definition applied to GODDAG. Here, two components d_1 and d_2 of a DXD are shown. Node p in component d_1 has content that subsumes completely the content of node s from component d_2. By applying the definition above, we can state that s is a *descendant* of p *in the given DXD*. Similarly, because the *ancestor* relationship is the inverse of the descendant, we can use the same idea to state that p is an *ancestor* of s in the DXD.

We can use similar intuition to redefine *following* and *preceding* axes. Indeed, a node x follows a node y iff the entire PCDATA content of x is located *after* the entire PCDATA content of y in a DOM tree. This statement is, again, independent on the DOM tree structure (as opposed to the definition of the *following* axis, which relies on the document order), and therefore can be transferred to GODDAG, as illustrated in the Figure 4. Node w of component d_2 has content that lies *after* the content of node p, hence, we can state the w *follows* p and, conversely, p *precedes* w.

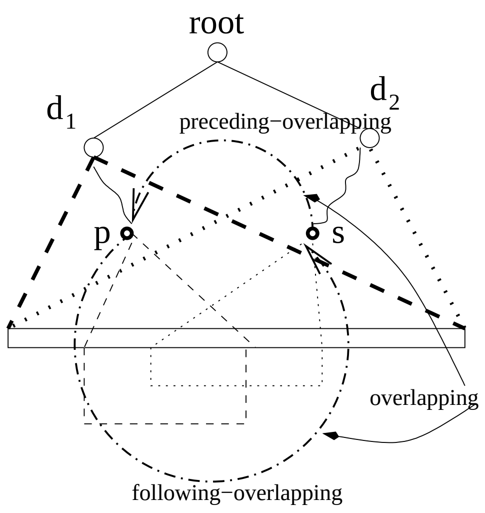


Figure 5: Axes for overlap in GODDAG.

At the same time, not all axes can be redefined in such a way, in particular, *child* and *parent* cannot transcend the boundaries of a single component in a way *descendant* and *ancestor* do. This is because unlike the notion of descendant, childhood-parenthood relations are tied to tree structures: a node x is a child of a node y iff there is an edge from y to x. Similarly, *preceding-sibling* and *following-sibling* axes rely on the existence of edges between the nodes in the DOM tree, not just on the position of the content. These axes, as well as *self* will not be extended beyond individual components of the distributed document.

One more observation can be made. Given a node x of a DOM tree, the five axes *ancestor*, *descendant*, *self*, *preceding* and *following* partition the entire DOM tree into five disjoint sets of nodes: that is, every node in the DOM tree will belong to exactly one of these axes as traversed from x. This property, however, does not hold in GODDAG: as shown in Section 2, there are GODDAG nodes with *overlapping* content (See Figure 5). Traversing the GODDAG using any of the five axes above will never yield any node that overlaps the context node in content. At the same time, as have been illustrated, queries over GODDAG require computing overlap. To accommodate for this need, we consider enhancing XPath with three new axes: *preceding-overlapping*, *following-overlapping* and *overlapping*. Intuitive meaning of these axes, as illustrated in Figure 5 is quite straightforward: x is in the result of applying *preceding-overlapping* axis to y iff x and y overlap in scope and x starts before y. In this case, y will be in the result of *following-overlapping* applied to x. The *overlapping* axis is the union of *preceding-overlapping* and *following-overlapping*.

We can now proceed to give formal definitions to XPath components over GODDAG, including the enhances apparatus to support markup overlap.

3.2 XPath over GODDAG

Let D be a distributed XML document over a concurrent XML hierarchy $C = \langle T_1, \ldots, T_k \rangle$. 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*. The first six axes are versions of the corresponding XPath axes extended to GODDAG. The remaining five axes do not have analogs in XPath.

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 use the relative positions of nodes in the GODDAG. However, we observe that there is no total document order over a GODDAG: overlapping markup will be incomparable.

DEFINITION 2. *The following new axes are defined:*

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 shown in Figure 3. (we use node labels to identify nodes in the graph)

(A) $xdescendant(21) = \{22, 24, 26, 23, 25, 27, 14, 16, 17, 19, 110, 13, 15, 18, 12, 112, 114, 113\}$. Note 21 corresponds to the `<page no="1">` markup. The *xdescendants* of this node are all its descendants in the "physical layout" component (lines 31, 32 and 33 and corresponding text nodes) as well as the contents of sentence 13 (nodes 12,13,15,18 and the corresponding text nodes). In addition, parts of sentence 14 (node 113 and text nodes 112 and 114) also are xdescendants of 21. At the same time, sentence 14 itself is *not* an xdescendant of page 1.

(B) $following{-}overlapping(26) = \{111, 115\}$
Node 26 represents `<line no="33">` markup from page 1. The scope of its content is leaf nodes L7—L10. The scope of node 111 (`<sentence no="14">` is L8—L15: because it starts after the scope of node 26 starts and ends after the scope of node 26 ends, it belongs to the result of evaluation of the *following-overlapping* axis. Incidently, text node 115 also overlaps node 26 on the right, thus it is added to the result as well.

Remark. We note that Definition 2 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*.

Proposed axes allow us to express queries to multihierarchical (distributed) documents in a straightforward manner. Consider, for example, the following queries:

(Q1): Find all sentences completely located on page 2;
(Q2): Find all words located on two lines;
(Q3): Find all sentences completely or partially located on page 1 of the document, that contain the word "charges";
(Q4): Find all occurrences of the word "Constitution" after page 1.

Table 1 shows the path expressions for these queries.

The algorithms for evaluation of the newly defined axes are given in [5].

4. EXPERIMENTAL RESULTS

We have fully implemented in Java an extension of XPath language that includes all the axes described in Definition 2. We call this processor *GOXPath*. GOXPath is a *main-memory processor*: all queries are processed over the memory-resident GODDAG structure, without addressing persistent storage. In this section we describe our preliminary study of the efficiency of this processor.

We report the results of four tests. The goals of the experiments were: (a) to compare the evaluation of extended XPath axes over documents with 2, 3, 4, 5, and 6 hierarchies; (b) to compare evaluation of axes over multihierarchical documents with different sizes (ranging from 5,000 nodes up to 500,000 nodes); (c) to compare evaluation of queries of different lengths; (d) to compare GOXPath performance with the execution of equivalent XPath queries (in terms of number of nodes manipulated) by Xalan[5] and Dom4j[6] processors. We emphasize that the goal of part (d) was not to prove that GOXPath is faster than certain XPath processors. Rather we want to show that on similar workloads GOXPath exhibits comparable performance. The tests were run on a Dell GX240 PC with 1.4Ghz Pentium 4 processor and 256 Mb main memory. The data input for the first two experiments was a distributed XML document obtained by multiplying the XML samples shown in Figure 2 enhanced with more markup when more than two hierarchies were used. The documents sizes ranged from 2MB up to 40MB on disk. Each query was evaluated four times, the average time was plotted.

In the first experiment we used 5 documents with 2, 3, 4, 5, and 6 hierarchies and of approximately 50, 55, 60, 65, and 70 thousand of nodes respectively. On these document instances we evaluated two queries, `/descendant::page/xdescendant::*`, and `/descendant::page/overlapping::*`, and the graphs of running time are shown in Figure 6 (a). The experimental results suggest that GOXPath performances are not significantly influenced by the number of hierarchies, but rather by the number of nodes that are manipulated (in the case of *xdescendant* axis the number of nodes slightly increases with the number of hierarchies, whereas for *overlapping* the number of nodes is approximately the same).

In the second experiment we studied the evaluation of extended axes for documents of different size (we used two hierarchies in this experiment). We ran two queries, `/descendant::page/xdescendant::*` and `/descendant::page/overlapping::*`, on documents of 5 up to 5,000 thousand nodes. The experimental results in Figure 6 (b) suggest linear dependence of axis evaluation on document size [5].

In the third experiment we tested GOXPath performance on queries of different length. We used two sets of eight queries each. Each query in the first set had the prefix `/descendant::page//`, and continued with 1 up to 8 `overlapping::*` location steps. Similarly, each query in the second set had a prefix `/descendant::page/`, and continued with 1, up to 8 `xdescendant-or-self::*` location steps (note that approximately the same number of nodes were

[5]http://xml.apache.org/xalan-j/
[6]http://www.dom4j.org

Query	Path Expression
Q1	`/xdescendant::page[@no="2"]/xdescendant::sentence`
Q2	`/xdescendant::word[overlapping::line]`
Q3	`/xdescendant::page[@no="1"]/xdescendant-or-overlapping::sentence[descendant::w[string(.)="charges"]]`
Q4	`/xdescendant::page[@no="1"]/xfollowing::w[string(.)="Constitution"]`

Table 1: Using newly defined axes to express queries over multihierarchical XML documents.

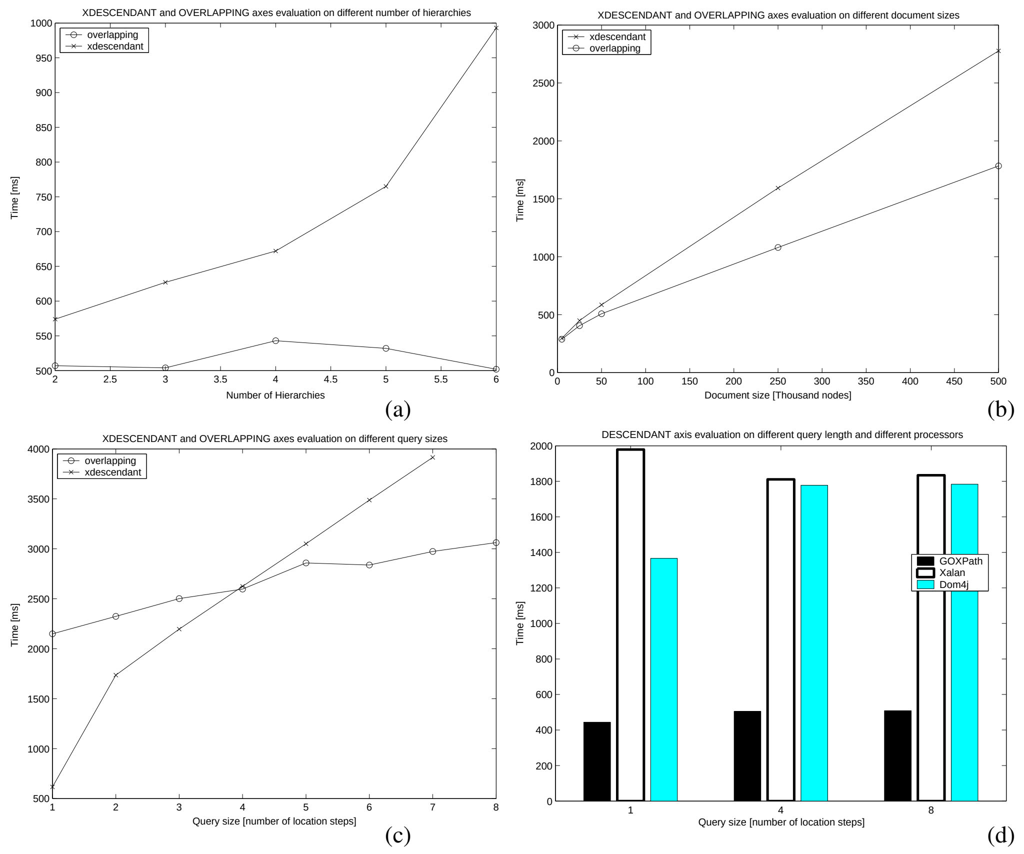


Figure 6: Experimental results.

processed at each step). The results shown in Figure 6 (c) clearly indicate linear dependence of query evaluation time on query size.

Finally, the last experiment compared the running time of GOXPath and XPath processors (Xalan and Dom4j) on workloads of comparable size (similar queries and the same number of nodes to be processed). We used documents with approximately 50 thousand nodes, two hierarchies for GOXPath, and markup fragmentation for Xalan and Dom4j. The same three queries were evaluated by each processor: `/descendant::page/descendant::*`, and the same as the preceding but ending with four, respectively eight `descendant::*` location steps. The test results are shown in Figure 6 (d) and demonstrate that GOXPath has similar performances as Xalan and Dom4j.

The experiments conducted here are preliminary and a more extensive testing is currently underway. But even these experiments show that our XPath implementation over GODDAG is efficient enough to be used in practice (and in fact, it is used as part of a larger suite of tools)[4].

5. REFERENCES

[1] XML Path Language (XPath) (Version 1.0). http://www.w3.org/TR/xpath, Nov 1999.

[2] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of PODS, San Diego, CA.*, pages 179–190, June 2003.

[3] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space eficiency. In *Proceedings of ICDE'03, Bangalore, India.*, pages 379–390, Mar 2003.

[4] I. E. Iacob and A. Dekhtyar. A framework for processing complex document-centric XML with overlapping structures. In *ACM SIGMOD Conference*, 2005. Demo, accepted.

[5] I. E. Iacob and A. Dekhtyar. Queries over Overlapping XML Structures. Technical Report TR 374-05, U. of Kentucky, CS Dept., March 2005. http://dblab.csr.uky.edu/∼eiaco0/publications/TR374-05.pdf.

[6] I. E. Iacob, A. Dekhtyar, and K. Kaneko. Parsing Concurrent XML. In *Proceedings WIDM*, pages 23–30, November 2004.

[7] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: one hierarchy isn't enough. In *Proceedings SIGMOD*, pages 251–262. ACM Press, 2004.

[8] M. M. Rabb. The civil rights program - letter and statement by the attourney general. The Dwight D. Eisenhower Library, Abilene, KS, http://www.eisenhower.utexas.edu/dl/Civil_Rights_Civil_Rights_Act/CivilRightsActfiles.html, April 10 1956.

[9] C. M. Sperberg-McQueen and C. Huitfeldt. GODDAG: A Data Structure for Overlapping Hierarchies. In *DDEP/PODDP, Munich*, pages 139–160, Sept. 2000.