IXPN: An Index-Based XPath Implementation

Gonzalo Navarro
Center for Web Research, Dept. of Computer Science, University of Chile
and
Manuel Ortega
Dept. of Computer Science, University of Chile

We present IXPN, and indexing technique for XML collections that permits fast processing of XPath queries. The underlying idea of IXPN is to map the XML/XPath model onto the Proximal Nodes (PN) model [Baeza-Yates and Navarro, ACM TOIS 1997], for which efficient algorithms have been designed. The mapping of XPath onto the query language of PN is rather involved because they are based on different concepts, but it can be done efficiently. On the side of the implementation of the PN model, we have completely reimplemented the 1997 prototype, and have added new operations needed to support XPath without disturbing the basic PN philosophy. In this paper we explain how the model mapping is done, how we have implemented the PN model, and how our implementation compares favourably against all the freely available alternatives we are aware of.

1. INTRODUCTION

There is little doubt that XML [Goldfarb and Prescod 1998] is bound to play an important role in the area of handling semistructured data. XML permits expressing the content and structure of a document, so that it can be read by a human and at the same time manipulated automatically, keeping maximum flexibility in the kind of structure that documents may have. XML is becoming a standard for manipulating, exchanging and storing semistructured data.

One of the most important operations needed on these "structured text" collections is that of searching for some piece of the collection that has some property. This property can be related to the text content and also to the structure. XPath [Consortium 1999a] is one of the most popular languages to query XML data. Although it has existed for several years, no fully satisfactory implementation of XPath exists, to the best of our knowledge.

Funded by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.
Authors' address: Gonzalo Navarro, Dept. of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Manuel Ortega, Dept. of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile. mortega@dcc.uchile.cl.

On the other hand, several theoretical models (in the sense of not being tied to any popular format such as XML) have been proposed in the last decades to query and manipulate structured text. One of those, called Proximal Nodes (PN) [Navarro and Baeza-Yates 1997], was designed with the aim of balancing expresiveness and implementation efficiency.

Recently, it has been shown that XQL [Lapp et al. 1998], a query language simpler than XPath and now less popular, could be mapped onto PN [Baeza-Yates and Navarro 2000]. No implementation was presented, however.

In this paper we tackle the problem of implementing the more powerful and popular XPath by mapping it onto the PN model. This mapping is not straightforward, because the design conceptions of XPath and PN are widely different. However, it can be done without loss of efficiency and we show carefully how this is carried out. Once transformed into a PN query, we find that the operations considered for the PN model in the original paper [Navarro and Baeza-Yates 1997] have to be changed slightly, some can be simplified, and others have to be added. We reimplement completely the PN model with a more efficient design. We show that all the operations can be implemented in time linear with the size of the arguments (and not of the database), and usually using very little main memory. At the end, we show how our prototype compares against existing freely available search engines for XPath.

The page of the IXPN prototype is `www.dcc.uchile.cl/ixpn`.

2. XML AND XPATH

We present in this section the XML and XPath specifications, in the depth necessary to understand how we implement XPath.

2.1 XML: eXtensible Markup Language

XML [Goldfarb and Prescod 1998] is the specification of a flexible markup language for structured text. The markup is expressed by means of special marks, called *tags*, that are inserted into the text in order to describe a structure. An XML tag is a sequence of characters between the special characters "<" and ">". All the text inside tags is part of the document structure, the rest is the content. Tags are human-readable, but are usually hidden when displaying the document, as they are not content but indicate how the content should be understood and presented. Figure 1 shows our running example of an XML document.

Tags are usually paired, so that portions of the content are marked by enclosing them between an initial and a final tag, typically `<tagname>` and `</tagname>`. For example,

```
<title>Introduction</title>
```

marks the text "Introduction" with the tag "title". When the enclosed content is empty, the initial and final tags are merged into one, `<tagname/>`. Apart from a name, tags may have attributes, each of which has a value. For example,

```
<image source="graph.png" caption="This is a graph"/>
```

is a tag without content and with two attributes, "source" and "caption", whose contents are "graph.png" and "This is a graph", respectively.

```
 1.   <?xml version="1.0" encoding="iso-8859-1"?>
 2.   <!DOCTYPE book "http://www.example.com/project.dtd">
 3.   <!--THE BOOK-->
 4.   <book>
 5.     <author>J. Williams</author>
 6.     <title>XCD Algorithm</title>
 7.     <chapter number="1">
 8.       <title>Introduction</title>
 9.       This is the Introduction...
10.       <image source="graph.png" caption="This is a Graph"/>
11.     </chapter>
12.     <chapter number="2">
13.       <title>Prototype</title>
14.       The prototype...
15.       <section number="2.1">
16.         <title>Implementation</title>
17.         <section number="2.1.1">
18.           <title>Data Model</title>
19.           Our data model...
20.         </section>
21.       </section>
22.     </chapter>
23.     <chapter number="3">
24.       <title>Conclusions And Future Work</title>
25.     </chapter>
26.   </book>
```

Fig. 1. Our running example of an XML document. Line numbers are not part of the document but used for future references.

An important aspect of XML is that tags cannot overlap, that is, a tag cannot be closed until all the contained tags have been closed. This induces a hierarchical structure on the document, where each node represents a tag, whose children are its attributes and contained tags. Figure 2 illustrates the hierarchy.

Each tag contains a text segment of the document, which in turn can contain more tags. Since tags have a length greater than zero, no two segments of different tags can start at the same position. Note also that the order in which initial tags are found in the text corresponds to a preorder traversal in the tree. This is known as the *document order*.

Apart from the tags that describe the structure, there are other elements in the XML specification that do not describe structure, such as comments, commands directed to specific processors, document type definitions, and so on. These are usually ignored in query languages and we ignore them in this paper.

2.2 XPath: XML Path Language

XPath [Consortium 1999a] is one of the favorite languages to select parts of an XML collection. It is an essential piece of more complete languages such as XQuery [Consortium 2001c] and XSLT [Consortium 1999b]. XPath has a full notation and an abbreviated version for the most commonly used operations. The result of an XPath query is a well-formed XML document.

XPath is composed of two parts. The first, most important for us, comprises

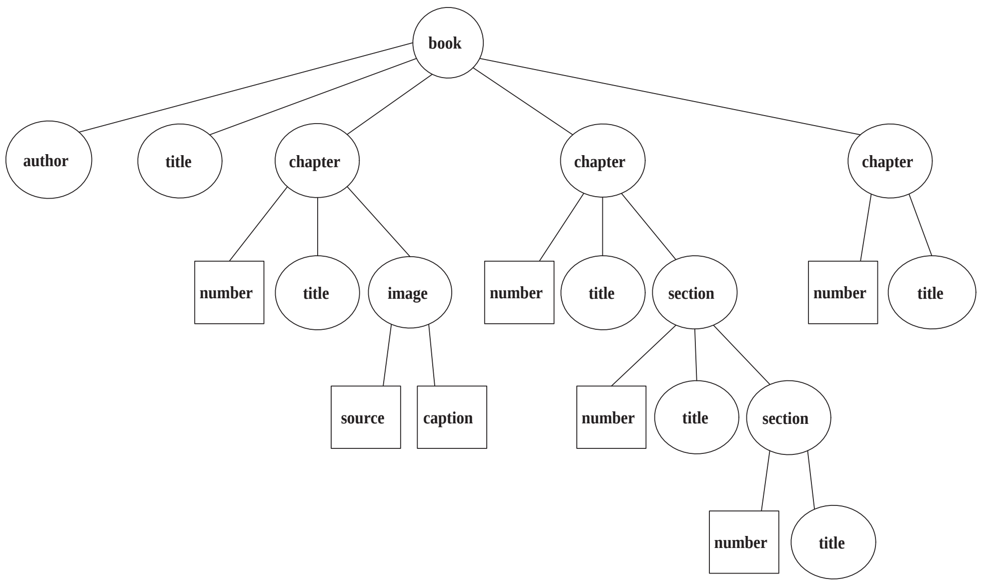


Fig. 2. The hierarchical structure of our running example. Circles represent tags and squares represent attributes.

the node selection operations, which let the user specify which parts of the XML collection to obtain. The second includes elements of a classical programming language, such as variables, expressions, and conditional and branching instructions. Indeed, XPath must be embedded into another language, from where variables and expressions take their semantics. In particular, for XPath embedded in XQuery, the so-called FLOWER expressions are permitted. These refer to For-Let-Where-Return structures, which permit specifying queries in XPath and present the results as desired in XML, perhaps forming structures not originally present in the document, much like SQL works on relational databases.

Figure 3 shows an example. It is clear that node selection operations are as important as the programming-like operations. However, from an implementation point of view, the former are much more challenging. Once the node selection operation returns the consecutive values for `$p`, the rest of the processing is rather simple: `$b` is obtained by variable substitution, the `WHERE` clause implies simple increments and comparisons, and so on.

```
FOR $p IN document("book.xml")//author
LET $b := document("book.xml")//book[author = $p]
WHERE count($b) > 100
RETURN $p
```

Fig. 3. An XQuery program using XPath in its first line.

This is why in this paper we focus only on the node selection operations of XPath: This is the most critical part of an implementation. It can be done very efficiently or very naively, as its language is very high-level. Its performance drives the overall performance. The programming language part, on the other hand, is rather low-level and therefore it can hardly experiment too large differences in implementation performance. In the example of Figure 3, the most challenging part is how are we going to find the `author` field across all the collection. The rest is fast and simple.

2.3 XPath Specifications

XPath distinguishes several *node types* in an XML document.

—An XML document is seen by XPath as rooted by a special and virtual *root-node*.

—All the XML tags that describe structure define *element-nodes*. These are the most common nodes.

—Each attribute of each tag makes up an *attribute-node*, which belongs to the corresponding element-node but is not considered to be a child of it.

—Other types of elements that we disregard in this paper because they are not used when implementing the basic XPath machinery are *namespace-nodes*, *instruction-nodes*, *text-nodes* and *comment-nodes*. In fact, text-nodes are of importance as they contain all the text content, but the concept of having virtual nodes that contain maximal text pieces is of no use for our translation. We treat the text in a different way.

The basic syntax of XPath consists of expressions, whose result is usually a set of nodes, but it can also be a boolean, numeric or string value. An expression specifies a set of nodes and optionally a function of the result. Hence it is possible to search for all the `section` nodes and just deliver the set, or add a counting operation and deliver instead the number of such nodes.

The mechanism used by XPath to describe the nodes that should be returned consists of four important parts: a *context*, an *axis*, a *nodetext*, and a *predicate*. In a first approximation, we can consider that all the nodes are considered as suitable context nodes, the axis specifies how to reach the selected nodes from the context nodes, the nodetest checks the name/type of the nodes to return, and the predicate further filters the desired selected nodes. For example, an XPath expression like

```
child::chapter[position()=1]
```

specifies a `child` axis, meaning that we want all the nodes that are children of some node (hence their parent is the context node). Furthermore, it specifies two predicates over the desired nodes: (1) a test of name: the node should be named `chapter`; (2) an explicit predicate (whose language we examine later): the position of the node should be the first in its context. This means that we want the first chapters of nodes found in the collection. In our running example, the full chapter number 1 (lines 7–11) is retrieved.

2.3.1 *Location Steps and Location Paths.*

The above is an example of a so-called *location step*. It always refers to some context node (that is outside its specification), and includes three parts:

(1) An axis specifying how to move from the context node to find the selected node (this can be `self` if we want to select the context node itself);

(2) a node test that checks the name of the selected node; and
(3) a predicate with zero or more expressions that further refine the selected nodes.

If a single location step is given alone, then all the nodes of the document are suitable context nodes, as in our previous example.

Location steps can be chained together to form a *location path*, which is a sequence of location steps where the result of each step becomes the set of context nodes for the next. This can be seen as the definition of a path in the structure tree that shows how answer nodes should be reached from context nodes and what conditions should context and selected nodes satisfy.

Consecutive location steps of a location path are separated by a "`/`". The context of the first location step is the set of all the nodes, and this location path is said to be "relative" (in the sense that the sequence of steps can appear anywhere in the tree). On the opposite, if a location path starts with a "`/`", this means that the context of the first location step is only the root node, and hence the path must be found only starting at the root. This path is called "absolute".

For example, an XPath expression like

```
self::chapter/child::section/child::title
```

is a location path formed by three location steps. It selects all the nodes of name `title` that are children of nodes of name `section` that are children of nodes of name `chapter`. In our running example it would return only the node

```
<title>Implementation</title>.
```

Figure 4 illustrates. Indeed, `child` is the default axis after a "`/`", and `self` is the first axis by default[1]. Hence the expression could be written simply as

```
chapter/section/title.
```

2.3.2 *Axes.* Let us now consider the axes in detail. Possible axes are:

`child:` direct children of the context node in the tree;
`descendant:` children, their children, and so on;
`parent:` tree parent of context node;
`ancestor:` parent, its parent, and so on;
`following-sibling:` siblings of the context node to its right;
`preceding-sibling:` siblings to the left;
`following:` nodes following the context node in document order, inside the same document and excluding descendants;
`preceding:` idem preceding the context node;
`attribute:` attributes of the context node;
`self:` the same context node;
`descendant-or-self:` self plus descendants; and
`ancestor-or-self:` self plus ancestors.

[1]In fact it could be said that `child` is always the default axis, and this makes no difference because every node has a parent except the root node, which is not retrievable.

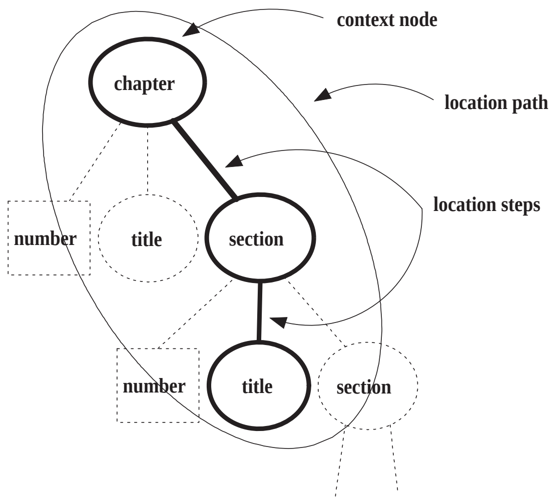


Fig. 4. A location path formed by three location steps.

`namespace:` this axis that is irrelevant for this paper.

Except for *attribute*, attribute nodes are never selected by these axes. Also, the result of these axes when the context is an attribute node is the empty set, except for *self*.

Note that the axes *ancestor*, *descendant*, *following*, *preceding*, and *self* partition the document tree into disjoint subsets. Attribute nodes are special nodes and considered as orthogonal to the rest of the model (as well as namespace nodes). Figure 5 illustrates.

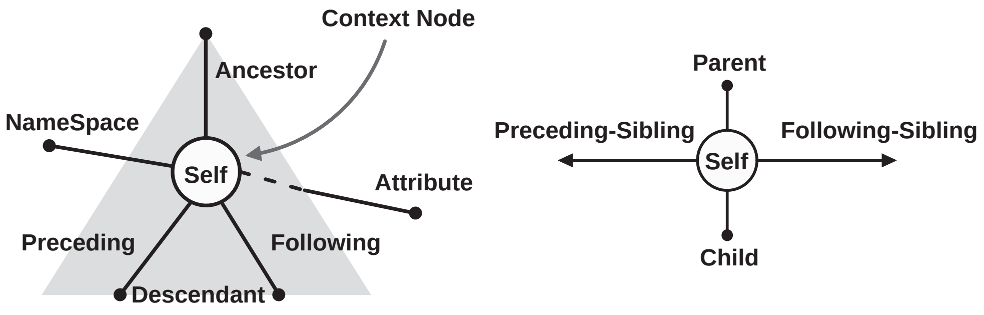


Fig. 5. The different axes and how they partition the document.

It is also important to mention that axes are classified into *reverse-axes* and *forward-axes*, depending on whether they take nodes that, in document order, are before or after the context node. Reverse axes are *ancestor*, *ancestor-or-self*, *preceding*, *preceding-or-self*, and *parent*[2]. All the rest are forward axes.

2.3.3 *Nodetests.* With respect to the test over the node, both the main node type (*element-node*, *attribute-node* or *namespace-node*) and the node name must match the test. The node type is given by the axis (*attribute* leads to an *attribute-node*, *namespace* to an *namespace-node*, and all the rest to an *element-node*). The term `node()` or the symbol "`*`" can be used to select any node name. Note that the *root-node* is never retrieved by an XPath query.

2.3.4 *Predicates.* Finally, let us consider the possible predicate filters. The whole set of possible expressions includes those defined in the embedding language, but there are some basic ones that are part of XPath. The predicates contain one or more basic conditions connected by *or*, *and* and *not*.

The simplest possible predicate is just a location path. The predicate becomes true if there exists such a location path in the context of the candidate node (to which the predicate applies). For example,

```
chapter[section/title]  =  self::chapter[child::section/child::title]
```

selects nodes of name `chapter` that satisfy the predicate of having a `title` child of a `section` as their child. The answer to this query in our running example is lines 12–22 (chapter 2).

An equality test, "`=`", compares either numbers or strings. In the case of strings, one operand must be a constant string and the other a location path. The test becomes true if the location path appears and its text content equals the string. A string containment test, "`=~`", is similar but the string should be contained in the text content of the location path[3]. As an example,

```
self::section[attribute::number="2.1.1"]
```

selects sections containing an attribute named `number` whose text content is exactly "2.1.1", that is, lines 17–20 of our running example. The example can be abbreviated as

```
section[@number="2.1.1"].
```

We remark that the text in attribute values is not considered to be a part of the text in other containing nodes.

The equality between numbers makes sense when we use some functions provided by XPath. These include, at least: `position()`, which is the position of the node (in document order) among those returned in the same context; `last()`, which is the number of nodes returned from this context. For example,

```
self::chapter/descendant-or-self::section[position()=last()]
```

gives the last section of each chapter, that is, lines 17–20 in our running example.

[2]According to the definition, *parent* could be classified either as forward or reverse axis. In our case it is simpler to see it as a reverse axis.

[3]We have used this operation for brevity. In rigor, this is written as "`contains(path,string)`" in XPath 1.0.

The example can be abbreviated as

`chapter//section[position()=last()]`.

Furthermore, a simple number is taken as a numeric equality over `position()`, so the example can be further abbreviated as

`chapter//section[last()]`.

Other abbreviations are "`.`", which stands for "`self::node()`" and "`..`", which stands for "`parent::node()`". Figure 6 summarizes the syntax of XPath, without abbreviations. It is a simplification of the official XPath 1.0 grammar.

```
     path  ⟶  / path | path / axis :: step | axis :: step
     axis  ⟶  child | descendant | descendant-or-self | parent
           |  ancestor | ancestor-or-self | following
           |  following-sibling | preceding | preceding-sibling
           |  attribute | namespace | self
     step  ⟶  nodetest | step [ pred ]
 nodetest  ⟶  NAME | node()
     pred  ⟶  pred and pred | pred or pred | not pred
           |  spath | numeric = numeric
    spath  ⟶  axis :: nodetest / spath | axis :: nodetest
           |  axis :: nodetest = string | axis :: nodetest =~ string
  numeric  ⟶  last() | position() | NUMBER
   string  ⟶  WORD | string WORD
```

Fig. 6. Summary of the syntax of XPath, abbreviations excluded.

3. PROXIMAL NODES

The Proximal Nodes Model (PN) [Navarro and Baeza-Yates 1995; Navarro and Baeza-Yates 1997] presents a good compromise between expressiveness and efficiency. It does not define a specific language, but a model in which it is shown that a number of useful operators can be included, while achieving good efficiency. Many independent structures can be defined on the same text, each one being a strict hierarchy, and allowing overlaps between areas delimited by different hierarchies (e.g. chapters/sections and pages/lines). A query can relate different hierarchies, but returns a subset of the nodes of one hierarchy only (i.e., nested elements are allowed in the answers, but not overlaps). Each node has an associated segment, which is the area of the text it comprises. The segment of a node includes that of its descendants. Text matching queries are modeled as returning nodes from a special "text hierarchy".

The model specifies a fully compositional language with three types of operators: (1) text pattern-matching; (2) to retrieve structural components by name (e.g. all chapters); and (3) to combine other results. The main idea behind the efficient evaluation of these operations is a bottom-up approach, by first searching the queries on contents and then going up the structural part. Two indices are used, for text and for structure, meant to efficiently solve queries of type 1 and 2 without traversing the whole database. To make operations of type 3 efficient, only operations that relate "nearby" nodes are allowed. Nearby nodes are those whose segments are more or less proximal. This way, the answer is built by traversing both operands in synchronization, leading in most cases to a constant amortized cost per processed element.

As we show next, many useful operators fit into this model. There is a separate text matching sublanguage, which is independent of the model. This model can be efficiently implemented, needing linear time for most operations and in all practical cases (this is supported by analysis and experimental results [Navarro 1995]). The time to solve a query is proportional to the sum of the sizes of the intermediate results (and not to the size of the database).

3.1 Query Language

The PN model permits any operation in which the fact that a node belongs or not to the final result can be determined by the identity and text position of itself and of nodes (in the operands) which are "proximal" to it, as explained.

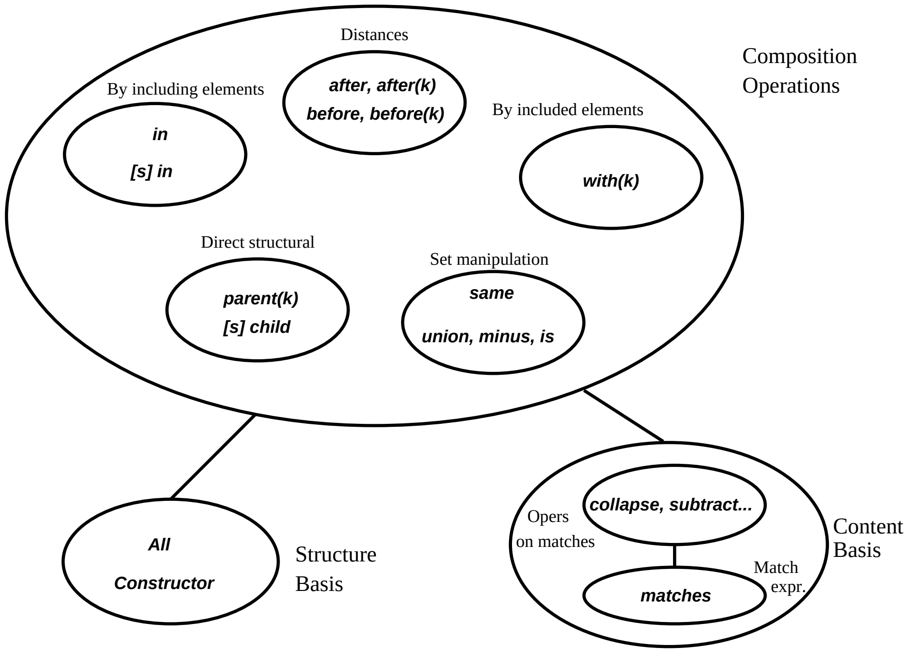


Fig. 7. Possible operations for the PN model, classified by type. We have removed those that are relevant when several hierarchies exist, which is not the case in XML.

Figure 7 shows the scheme of a possible set of operations. There are basic extraction operators (forming the basis of querying on structure and on contents), and operators to combine results from others, which are classified in a number of groups: those which operate by considering included elements, including elements, nearby elements, by manipulating sets and by direct structural relationships.

We explain in some detail those that are relevant for the case of a single hierarchy, which includes the XML model.

—Matching sublanguage: Is the only one which accesses the text content of the database, and is orthogonal to the rest of the language.
  —Matches: The matching language generates a set of disjoint segments, which are introduced in the model as belonging to a special "text hierarchy". All the text answers generate flat lists. For example, `"Introduction"` generates the flat set of all segments of 12 letters where that word appears in the text (those are contained in lines 8 and 9 of our running example). Note that the matching language could allow much more complex expressions (e.g. regular expressions).
  —Operations on matches: Are applicable only to subsets of the text hierarchy, and make transformations to the segments. We see this point and the previous one as the mechanism for generating match queries, and we do not restrict our language to any sublanguage for this. As an example, M **collapse** M' superimposes both sets of matches, merging them when an overlap results; and M **subtract** M' removes from the first set the text positions belonging to the second set, shortening, removing and cutting segments as required.

—Basic structure operators: Are the other kind of leaves of the query syntax tree, which refer to basic structural components.
  —Name of structural component: ("constructor" queries). Is the set of all nodes of the given type. For example, `chapter` retrieves all the chapter elements (3 nodes in Figure 2).
  —Whole hierarchy: ("**All**" queries). Is the set of all nodes of the hierarchy. The same effect can be obtained by summing up ("**union**" operator) all the node types of the hierarchy.

—Included-In operators: Select elements from the first operand which are included in one of the second.
  —Free inclusion: Select any included element. "P **in** Q" is the set of nodes of P which are included in a node of Q. For example, `title` **in** `chapter` selects all titles inside chapters, even section titles (see Figure 2).
  —Positional inclusion: Select only those elements included at a given position. In order to define position, only the top-level included elements for each including node are considered. "[s] P **in** Q" is the same as **in**, but only qualifying the nodes which descend from a Q-node in a position (from left to right) considered in s. The language for expressing positions (i.e. values for s) is also independent. It was considered that finite unions of $i..j$, $last - i..last{-}j$, and $i..last{-}j$ would suffice for most purposes. The range of possible values is $1..last$. For example, `[1..2] chapter` **in** `book` retrieves the first two chapters from our book example. If chapters included other chapters, only the top-level ones would be considered.

—Including operators: Select from the first operand the elements including elements from the second one. "P **with**(k) Q" is the set of nodes of P which include at least k nodes of Q. If (k) is not present, we assume 1. For example, `chapter`

with(2) `"Introduction"` selects the chapters in which the word "Introduction" appears at least two times (chapter 1 in our example).

—Direct structure operators: Select elements from the first operand based on direct structural criteria, i.e. by direct parentship in the structure tree corresponding to the hierarchy.

—"[s] P **child** Q" is the set of nodes of P which are children (in the hierarchy) of some node of Q, at a position considered in s (that is, the s-th children). If [s] is not present, we assume $1..last$. For example, `title` **child** `chapter` retrieves the titles of all chapters (and not titles of sections inside chapters).

—"P **parent**(k) Q" is the set of nodes of P which are parents (in the hierarchy) of at least k nodes of Q. If (k) is not present, we assume 1. For example, `chapter` **parent**(3) `section` selects chapters with three or more top-level sections (none in our example).

—Distance operators: Select from the first operand elements which are at a given distance of some element of the second operand, under certain additional conditions.

—"P **after/before** Q (C)" is the set of nodes of P whose segments begin/end after/before the end/beginning of a segment in Q. If there is more than one P-candidate for a node of Q, the nearest one to the Q-node is considered (if they are at the same distance, then one of them includes the other and we select the including one). In order for a P-node to be considered a candidate for a Q-node, the minimal node of C containing them must be the same, or must not exist in both cases. For example, `image` **after** `title` (`chapter`) retrieves the nearest images following titles, inside the same chapter (the only image would be retrieved in our example).

—"P **after/before**(k) Q (C)" is the set of all nodes of P whose segments begin/end after/before the end/beginning of a segment in Q, at a distance of at most k text symbols (not only nearest ones). C plays the same role as above. For example, `"Conclusions"` **before** (20) `"Future"` (`chapter`) selects the words "Conclusions" that are followed by "Future" at a distance of at most 20 symbols, inside the same chapter (there is one occurrence in chapter 3 in our example).

—Set manipulation operators: Manipulate both operands as sets, implementing union, difference, and intersection under different criteria.

—"P **union** Q" is the union of P and Q. For example, `figure` **union** `list` is the set of all figures and lists. To make a union on text segments, one uses **collapse**.

—"P **minus** Q" is the set difference of P and Q. For example, `chapter` **minus** (`chapter` **with** `image`) are the chapters with no images (chapters 2 and 3 in our example). To subtract text segments, one resorts to operations on matches.

—"P **is** Q" is the intersection of P and Q. For example, ([1] `title` **in** `chapter`) **is** (`[3]` `title` **in** `book`) selects the titles which are first (top-level) title of a chapter and at the same time third (top-level) title of the book (the title of chapter 2 would be selected in our example). To intersect text segments use **same**.

—"P **same** Q" is the set of nodes of P whose segments are the same segment of a node in Q. For example, `title` **same** `"Introduction"` gets the titles that say (exactly) "Introduction". This gives the title of chapter 1 in our example.

Except for set manipulation ones, the model also permits the negated version of all the operators. For example, P **not with** Q is the same as $P - (P$ **with** $Q)$, although the evaluation is more efficient.

Clearly inclusion can be determined by the text area covered by a node, and the fact that an element in A qualifies or not depends only on elements of B that include it or are included in it. Direct ancestorship can be determined by the identity of the nodes and appropriate information on the hierarchical relations between nodes. Note that just the information on text areas covered is not enough to discern between direct and general inclusion. Distance operations can be carried out by just considering the areas covered and by examining nearby elements of the three operands. Finally, set manipulation needs nothing more than the identity of the nodes and depend on nearby nodes of the other operands.

3.2 Existing Implementation

The PN model proposes an implementation where an index is built on the structure of the text separated from the normal index for the text content. The structural index is basically the hierarchy tree with pointers to know the parent, first child and next sibling of each node. In addition, implicit lists (with "next sibling" and "first child" pointers) for each different structural element are maintained, so that one can traverse the complete tree or the subtree of all the nodes of a given type. Figure 8 illustrates.

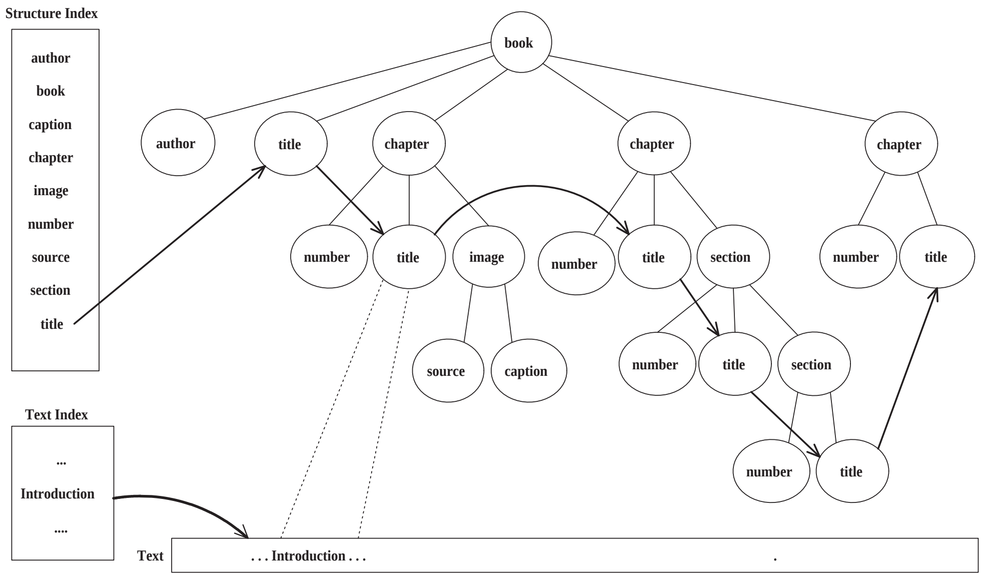


Fig. 8. Structure and text index over our XML document, with the links for `title` highlighted.

Two different implementations of the model are proposed. A *full evaluation* version solves the query syntax tree recursively, that is, both operands of the root are (recursively) solved completely and then the root operator is applied to both arguments, which are by this time fully evaluated. A *lazy evaluation* version regards the query syntax tree as an entity that survives across the whole evaluation, to which one requests results one by one. Upon receiving a request, any node of this syntax tree requests in turn results from its operand subtrees until it has enough information to deliver one result. In the experiments [Navarro 1995] the lazy version worked better for more complicated queries and worse for simpler queries.

Leaves which correspond to structural elements are solved by using the structure index directly; those which correspond to pure queries on the text content are solved with the classical index on content (e.g. an inverted file) and translated into a list of text segments that match the query. This list is treated as a particular case of a tree of answers.

The intermediate (and final) results are trees which are subsets of the whole hierarchy. Figure 9 illustrates.

As defined by the model, all the allowed operations can be solved by a synchronized linear traversal over the operands, so that the total time to solve a query is proportional to the total size of the intermediate results, usually linear time.

4. MAPPING XML/XPATH ONTO PROXIMAL NODES

In this section we describe how the XML/XPath model has been mapped onto the PN model. This mapping has two parts: mapping the data and mapping the operations.

4.1 Mapping the XML Structure

First of all, the PN model permits independent hiearchies, while XML has only one. This makes it possible to simplify the implementation of PN described in [Navarro and Baeza-Yates 1997; Navarro 1995]. The special "text hieararchy" defined in the PN model, however, has to be maintained in order to permit text searches.

An aspect where XML is more complex than PN is that XML permits different node types: Although most XML nodes are element-nodes, there are also attribute-nodes and other node types. In PN there exists a single node type.

We have circumvented this problem by considering all nodes as element-nodes. In those nodes that are not originally element-nodes, we add a special initial character to their name so that we can know which node types they were originally. This can be applied to all node types. However, we have done this only to attribute-nodes (to which we added the special character "`@`", although any other one would do).

In order to handle collections of XML documents, we have added a new node type called *document-node*. These are virtual (like the root-node) and not retrievable. We have translated them as if they corresponded to tag name "`-DOCUMENT`". Similarly, the root-node is named "`-ROOT`".

The other node types are not interesting for searching and thus can be disregarded.

For example, a tag with attributes like

```
<chapter number="1"> ...  </chapter>
```

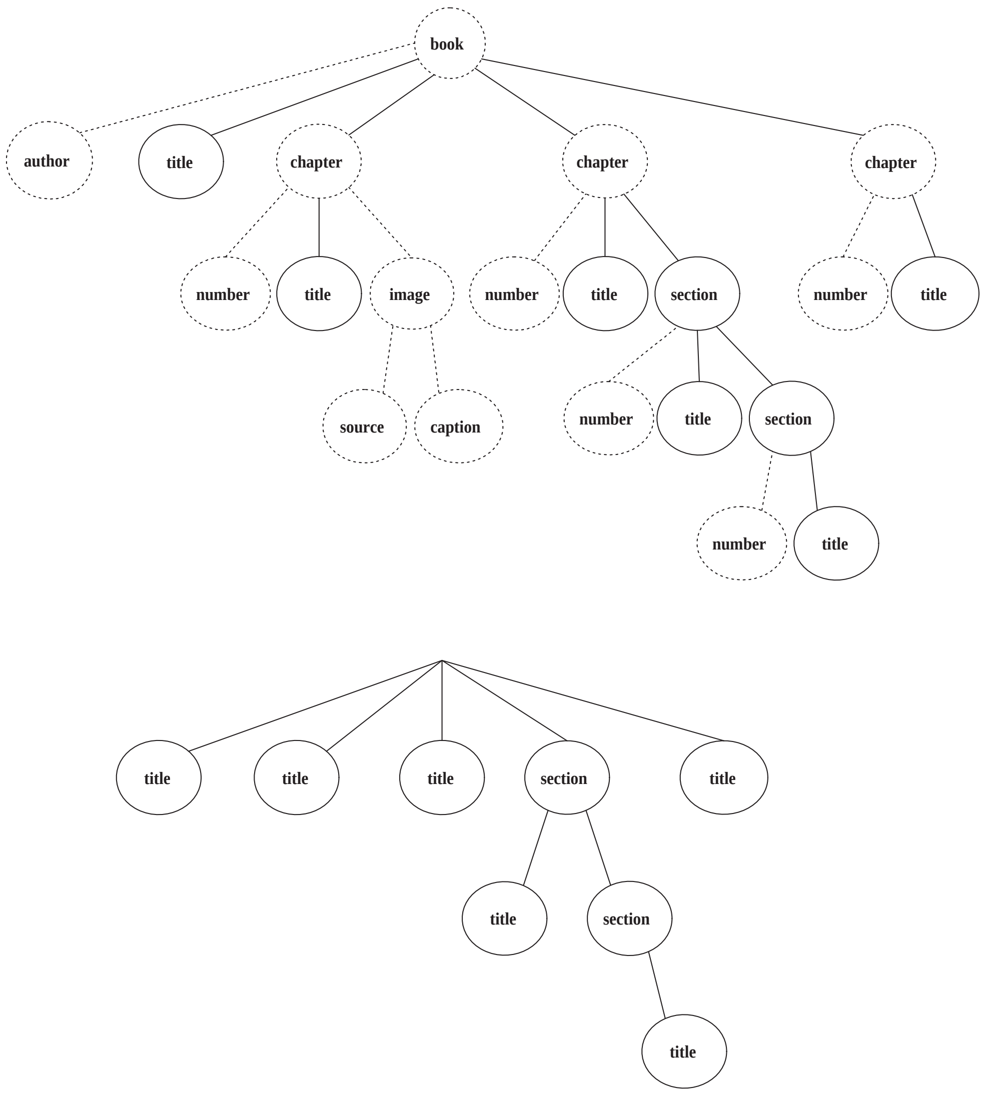


Fig. 9. Tree result of the query "`section` **union** `title`".

will be regarded as

```
<chapter><@number>1</@number> ...  </chapter>.
```

We still, however, refer to text positions in the original file. This requires some care when it comes to define which is exactly the text segment that corresponds to each node type, so that segments of parent nodes strictly contain those of child nodes. The rules are as follows:

(1) For element-nodes with start and end tags, the segment starts two positions after the last character of the initial tag name, and finishes at the position

preceding the closing tag;

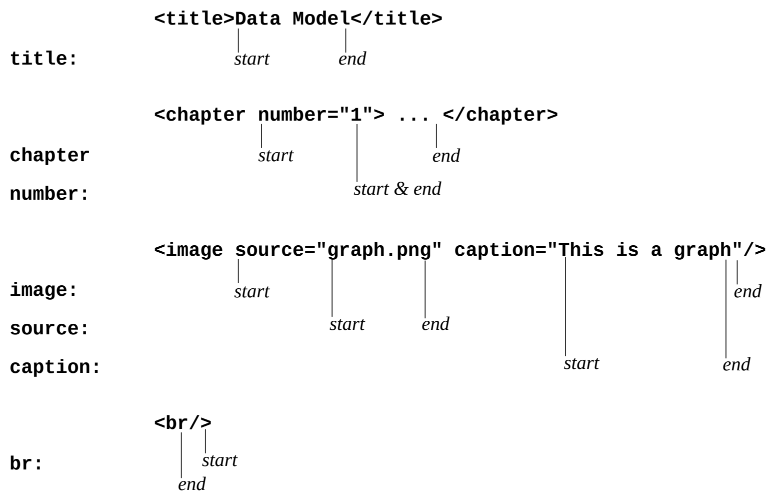
(2) for element-nodes with a single start/end tag, the segment starts with the same rule as for (1), and ends at the position preceding the final "/" (this may cause the finishing position to be before the starting position, but it causes no troubles, as in this case there cannot be children nor attributes);

(3) for attribute-nodes, the segments cover exactly the area of their attribute value, excluding quotation marks;

(4) for document-nodes, the segment goes from the first to the last characters of the document;

(5) for the root-node, the segment goes from a ficticious position 0 (zero) to one position after the last character of the last document (also ficticious).

Figure 10 illustrates some cases. This scheme preserves the document order of the nodes and is well defined for the XPath and PN operations, as explained. Additionally, it has the advantage of easing the displaying of results: If one knows that a given segment with a known tag name has matched the query, one can simply expand the segment by the tag name length plus a fixed amount in each direction in order to obtain a well-formed XML node to display. Finally, the property of strict segment containment simplifies several PN algorithms [Navarro and Baeza-Yates 1997; Navarro 1995].



<title>Data Model</title>
title:
start
end
<chapter number="1"> ... </chapter>
chapter
number:
start
end
start & end
<image source="graph.png" caption="This is a graph"/>
image:
source:
caption:
start
end
start
end
start
end
<br/>
br:
start
end

Fig. 10. Examples of segment coverage for XML tags.

In order to enforce that texts below attributes do not belong to other containing nodes, we state that words in attribute values should have added a blank character at their beginning, so they cannot be confused with words belonging to text-nodes. We see soon how this features is used.

We note that XML permits references between documents, which can be queried in languages like XLink [Consortium 2001a] and XPointer [Consortium 2001b], but not in XPath. For this reason we disregard these references when considering the structure of the collection. Implementing queries on these references, however, probably needs techniques that are well beyond the capabilities of an XPath implementation: The references induce an arbitrary graph structure in the text collection, not necessarily a hierarchy. Most of the efficiency of our XPath implementation strongly relies on a hierarchical structure.

4.2 Mapping XPath Expressions

Three aspects have to be taken into account when mapping XPath expressions onto PN: the contexts, the axes, and the predicates.

While XPath is strongly based on the notion of context, this concept does not exist in PN. Yet, the conversion is possible. XPath expressions are regarded as sequences of location steps, where the result of the current step makes up the context for the next. Previous and current location steps are related by the axes. PN expressions, on the other hand, can be seen as a composition of binary relations between node sets. The types of binary relations are quite similar to those denoted by the axes. Hence it is possible to convert sequences of location steps into a composition of binary relations. The operands of these relations are given by the node tests and the composition of the location path itself. Predicates, on the other hand, can similarly be translated into a composition of relations, as will be made clear soon.

4.2.1 *Nodetests.* The most common nodetest is just an element-node name. This is translated into PN simply as the same structural name. Note that, if the name starts with the special character "`@`", then it is indeed an attribute-node name, but we need not pay special attention to this fact.

The other possible nodetest is `node()` (abbreviated "`*`"), which corresponds to the set of all element-nodes. This is translated into a variant of the **All** operand of PN, namely **Node**. The abbreviation "`@*`" stands for all the attribute-nodes, and is translated into another new PN operand named **Attribute**. Both new operands will be implemented as variants of **All**.

4.2.2 *Axes.* Most axes of XPath have their counterpart in PN operations. Some PN operations, however, must be slightly redefined, and others have to be created from scratch. However, the new operations fit well in the philosophy of PN. Moreover, some axes that exist as PN operations can be simplified for an XML structure. We present now the axes of XPath and their PN counterparts.

`child:` corresponds to **child** operation in PN, where the $[s]$ modifier is not used. Note, for this item and the rest, that we plan to translate paths in reverse, for example `section/title` becomes "`title` **child** `section`". Note that if the abbreviation "`@name`" is used as a nodetest, then "`child`" should be understood as "`attribute`" (a later item in this list), and the special character "`@`" removed. The fact that "`@`" is used both in XPath and by ourselves to denote attribute nodes makes it possible to not doing anything special when dealing with this kind of names.

`parent:` corresponds to **parent** operation in PN, where the (k) modifier is not used.

`descendant:` corresponds to **in** operation in PN, without the $[s]$ modifier.

`descendant-or-self`: can be implemented in PN as "(P **is** Q) `union` (P **in** Q)". It is, however, much simpler and efficient to add a new **inself** operation to PN with the proper semantics.

`ancestor`: corresponds to **with** operation in PN, without the (k) modifier.

`ancestor-or-self`: again we choose to add a new operation **withself** to PN.

`following`: corresponds to **after**(∞)(`-DOCUMENT`) operation in PN. For brevity we call it just **after** (and make a special, faster and simplified, implementation for it).

`preceding`: similarly, it corresponds to **before**(∞)(`-DOCUMENT`) operation in PN, which we will call just **before**.

`following-sibling`: can be implemented in PN as "(P **after** Q) **child** (**Node parent** Q)". It is, however, much simpler and efficient to add a new **after-sibling** operation to PN with the proper semantics.

`preceding-sibling`: just as before, we add a new **before-sibling** operation to PN.

`attribute`: is similar to `child`, but also we enforce the selection of attribute-nodes only. It is implemented in PN by simply adding the special character "`@`" at the beginning of the nodetest, even if this nodetest is "`*`", and translate it as **child**.

`self`: corresponds to **is** operation in PN.

As explained, the translation of a location path is done in reverse. For example,

`chapter/section/title` = `self::chapter/child::section/child::title`,

would select all titles children of sections children of chapters. In our running example this is the content of line 16. The expression is translated into

`title` **child** (`section` **child** `chapter`).

Another example could be

`self::image/ancestor::chapter/following-sibling::chapter`,

which would select chapters that follow chapters (from the same book) that contain images (the whole chapters 2 and 3 in our running example). This expression would be translated into

`chapter` **after-sibling** (`chapter` **with** `image`).

Yet a third example, involving abbreviations, is "`//image`", which stands for

`/descendant-or-self::node()/child::image`,

and would be translated into

`image` **child** (**Node inself** `-ROOT`).

Both are indeed equivalent to just "`image`". Later we will give some simplification rules for the resulting PN expressions.

4.2.3 *Predicates.* Proximal Nodes has no concept of predicate. However, predicates can be translated into additional compositions with the PN algebra. It is important, however, that all predicates are solved as sets of nodes instead of boolean or numeric values, as these cannot be handled as intermediate values in PN.

The main idea applies to predicates that consist simply of a location path. This location path can be translated similarly as location paths outside predicates. This time, however, we must reverse the order and meaning of operands. Axes with opposite meaning are, for example, `child`↔`parent` and `descendant`↔`ancestor`.

By default, the first axis of the predicate is `child`. For example,

`chapter[section/title]` = `chapter[child::section/child::title]`,

which selects chapters that are parents of sections that are parents of titles, is translated into the PN expression

`chapter` **parent** (`section` **parent** `title`).

The default axis can be overwritten, for example using

`chapter[//title]` = `chapter[descendant-or-self::node()/child::title]`,

which selects chapters containing titles, is translated into the PN expression

`chapter` **withself** (**Node parent** `title`) = `chapter` **with** `title`,

where the second expression is obtained after algebraic simplification of the PN expression.

Yet a third example is

`image[@*]` = `image[attribute:*]`,

which selects images with attributes and is translated into the PN expression

`image` **parent Attribute**.

When a location path is compared against a string, the translation uses **with** for containment and **same** for equality. The operation is applied to the final step of the path. Phrases are translated using a new operation of PN called **phrase**. This operation belongs to the matching sublanguage, and is the only operation we need from that sublanguage. The basic matching, on the other hand, requires only searching for whole words.

For example,

`chapter[@number="1"]` = `chapter[attribute::number="1"]`,

chooses the chapter whose attribute "`@number`" has the string value "1". This is translated into PN as

`chapter` **parent** (`@number` **same** `" 1"`),

where we note that we have added a blank in front of the "1", as we are dealing with text inside attributes.

Similarly,

`section[title=~"Model"]` = `section[child::title=~"Model"]` chooses the sections whose titles contain the word "Model". This is translated into PN as

`section` **parent** (`title` **with** `"Model"`).

Let us now consider the boolean operations that can appear in predicates. The "`and`" operation can be solved just by composing the conditions, as these are naturally restricting the previous result. For example,

`image[@source and @caption]`

is translated into

(`image` **parent** `@source`) **parent** `@caption`.

The "`or`" operation, instead, requires explicit union of both results, is translated into the **union** operation in PN. For example,

`image[@source or @caption]`

is translated into

(`image` **parent** `@source`) **union** (`image` **parent** `@caption`).

Finally, the "`not`" operation can be applied to a whole path (or path with a final equality/containment test), denoting that the context node should not match such a location path. This could be easily translated using the set difference operator ("**minus**") of PN, although we choose a faster option: negated versions of all the PN operations are used to connect the context node and the predicate.

To conclude this section we must explain how we handle the numeric predicates, which may specify that only some qualifying nodes must be returned, namely those at specific positions (in document order) within the set of qualifying nodes for each context node. These are solved by first obtaining all the answers and later choosing the appropriate positions. In case the positions do not mention `last()`, it may not be necessary to generate all the answers. For example, the expression

`chapter[2]` = `chapter[position()=2]`

requires obtaining only the second chapter. Node that this resembles the $[s]$ modifier of **child** and **in**, but this time we need it applied to every possible axis. Hence we need a general, independent method.

4.3 A Formalization

To summarize the whole method in a complete and unambiguous way, we present now a formalization of the transformation of XPath into PN expressions. This is expressed in terms of a transformation function $\mathcal{PN}$, which gives the PN expression equivalent to a given XPath expression.

We use some auxiliary functions: $\mathcal{A}$ transforms axes into PN operations, $\mathcal{A}^{\mathcal{R}}$ into reverse PN operations, and $\mathcal{A}^{\mathcal{RN}}$ into reverse and negated PN operations. On the other hand, $\mathcal{N}$ transforms nodetests into PN operands. Tables 1 and 2 define these auxiliary functions.

Before any translation we perform a conversion on `attribute` axes. This is as follows: any occurrence of the form

`.../attribute::nodetest...`

Axis	$\mathcal{A}$	$\mathcal{A}^{\mathcal{R}}$	$\mathcal{A}^{\mathcal{RN}}$
`child`	**child**	**parent**	**not parent**
`parent`	**parent**	**child**	**not child**
`descendant`	**in**	**with**	**not with**
`descendant-or-self`	**inself**	**withself**	**not withself**
`ancestor`	**with**	**in**	**not in**
`ancestor-or-self`	**withself**	**inself**	**not inself**
`following`	**after**	**before**	**not before**
`preceding`	**before**	**after**	**not after**
`following-sibling`	**after-sibling**	**before-sibling**	**not before-sibling**
`preceding-sibling`	**before-sibling**	**after-sibling**	**not after-sibling**
`self`	**is**	**is**	**not is**

Table 1. Formal translation of axes. $\mathcal{A}$ is used in normal location paths, while $\mathcal{A}^{\mathcal{R}}$ and $\mathcal{A}^{\mathcal{RN}}$ are used for location paths inside predicates. The latter is used to translate boolean negation. We do not specify how to translate the `attribute` axis because we never let that case occur.

Nodetest	$\mathcal{N}$
name	*name*
`@`*name*	`@`*name*
`node()`	**Node**
`*`	**Node**
`@*`	**Attribute**

Table 2. Formal translation of nodetests.

is converted into

`.../child::@nodetest...`

that is, an "`@`" is added at the beginning of the nodetest and the `attribute` axis becomes `child`. Moreover, if a string comparison follows the nodetest, all their words get added a blank before their first character.

Our translation follows, based on the abstract unabbreviated syntax of Figure 6. The first rule that matches an argument is the one used. We remark that we keep translation rules as simple as possible, and deal later with possible inefficiencies incurred.

We translate location paths by always considering its last element first. Our first rule translates absolute paths into relative paths. The second rule specifies how location paths are split into location steps. The third rule shows how the sequence is finished.

$$\begin{aligned}
\mathcal{PN}(\texttt{/}path) &= \mathcal{PN}(\texttt{self::-ROOT/}path)\\
\mathcal{PN}(path\texttt{/}axis\texttt{::}step) &= (\mathcal{S}(step)\ \mathcal{A}(axis)\ \mathcal{PN}(path))\\
\mathcal{PN}(axis\texttt{::}step) &= (\mathcal{S}(step)\ \mathcal{A}(axis)\ \textbf{Node})
\end{aligned}$$

Function $\mathcal{S}$ specifies how to translate a single location step, axis excluded. The first rule translates a simple nodetest, while the second rule handles consecutive predicates by resorting to a function $\mathcal{R}$.

$$\begin{aligned}
\mathcal{S}(nodetest) &= \mathcal{N}(nodetest)\\
\mathcal{S}(step\texttt{[}pred\texttt{]}) &= \mathcal{R}(\mathcal{S}(step), pred)
\end{aligned}$$

Let us now consider $\mathcal{R}$, which translates predicates. The idea is that the first argument of $\mathcal{R}$ is the context (already a PN expression) and the second is the predicate. This time we consider the location paths from left to right, and reverse the axes. The first rule specifies how location paths are split into location steps. The second rule treats the case of a single location step. The third and fourth rules deal with string comparisons. The final six rules work out the boolean connectives. For the "`not`" connective, we assume that it is applied only to paths, otherwise the obvious boolean equivalences are applied. For brevity we have used *ntst* instead of *nodetest*.

$$\begin{aligned}
\mathcal{R}(ctx, axis\texttt{::}ntst\texttt{/}spath) &= (ctx\ \mathcal{A}^{\mathcal{R}}(axis)\ \mathcal{R}(\mathcal{N}(ntst), spath))\\
\mathcal{R}(ctx, axis\texttt{::}ntst) &= (ctx\ \mathcal{A}^{\mathcal{R}}(axis)\ \mathcal{N}(ntst))\\
\mathcal{R}(ctx, axis\texttt{::}ntst\texttt{=}string) &= (ctx\ \mathcal{A}^{\mathcal{R}}(axis)\ (\mathcal{N}(ntst)\ \textbf{same}\ \mathcal{P}(string)))\\
\mathcal{R}(ctx, axis\texttt{::}ntst\texttt{=\~{}}string) &= (ctx\ \mathcal{A}^{\mathcal{R}}(axis)\ (\mathcal{N}(ntst)\ \textbf{with}\ \mathcal{P}(string)))\\
\mathcal{R}(ctx, pred_1\ \texttt{and}\ pred_2) &= \mathcal{R}(\mathcal{R}(ctx, pred_1), pred_2)\\
\mathcal{R}(ctx, pred_1\ \texttt{or}\ pred_2) &= (\mathcal{R}(ctx, pred_1)\ \textbf{union}\ \mathcal{R}(ctx, pred_2))\\
\mathcal{R}(ctx, \texttt{not}\ axis\texttt{::}ntst\texttt{/}spath) &= (ctx\ \mathcal{A}^{\mathcal{R}\mathcal{N}}(axis)\ \mathcal{R}(\mathcal{N}(ntst), spath))\\
\mathcal{R}(ctx, \texttt{not}\ axis\texttt{::}ntst) &= (ctx\ \mathcal{A}^{\mathcal{R}\mathcal{N}}(axis)\ \mathcal{N}(ntst))\\
\mathcal{R}(ctx, \texttt{not}\ axis\texttt{::}ntst\texttt{=}string) &= (ctx\ \mathcal{A}^{\mathcal{R}\mathcal{N}}(axis)\ (\mathcal{N}(ntst)\ \textbf{same}\ \mathcal{P}(string)))\\
\mathcal{R}(ctx, \texttt{not}\ axis\texttt{::}ntst\texttt{=\~{}}string) &= (ctx\ \mathcal{A}^{\mathcal{R}\mathcal{N}}(axis)\ (\mathcal{N}(ntst)\ \textbf{with}\ \mathcal{P}(string)))
\end{aligned}$$

Numeric predicates are not included in the translation rules because they are not translated but implemented directly, as explained. Finally, function $\mathcal{P}$ translates phrases (sequences of words) into PN expressions.

$$\begin{aligned}
\mathcal{P}(word) &= word\\
\mathcal{P}(string\ word) &= (\mathcal{P}(string)\ \textbf{phrase}\ word)
\end{aligned}$$

4.3.1 *Algebraic Optimizations.* The above rules are designed to be as simple to understand as possible. However, they may generate unnecessarily complex PN expressions. Most of them can be simplified back by finding places where **Node** and **Attribute** are mentioned, and applying some simplification rules, as follows: All the expressions that follow are equivalent to just P.

$$\begin{array}{rl}
P\ \textbf{is Node} & \textbf{Node is}\ P\\
P\ \textbf{is Attribute} & \textbf{Attribute is}\ P\\
P\ \textbf{inself Node} & P\ \textbf{withself Node}
\end{array}$$

$$\begin{array}{rl} P\ \textbf{in Node} & P\ \textbf{child Node} \\ P\ \textbf{inself}\ \texttt{-ROOT} & P\ \textbf{in}\ \texttt{-ROOT} \end{array}$$

Note that the fourth line is valid because we are not interested in returning the `-ROOT` node, as it is ficticious. Other algebraic equivalences of interest are

$$P\ \textbf{withself}\ (\textbf{Node parent}\ Q) \ = \ P\ \textbf{parent}\ (\textbf{Node withself}\ Q) \ = \ P\ \textbf{with}\ Q$$

$$P\ \textbf{inself}\ (\textbf{Node child}\ Q) \ = \ P\ \textbf{child}\ (\textbf{Node inself}\ Q) \ = \ P\ \textbf{in}\ Q$$

$$P\ \textbf{is}\ (\textbf{Node op}\ Q) \ = \ (\textbf{Node op}\ Q)\ \textbf{is}\ P \ = \ P\ \textbf{op}\ Q$$

for any operation **op**.

Many other optimizations are possible, but those above fix the inefficiencies included when we automatically transform XPath into PN expressions.

5. IMPLEMENTING PROXIMAL NODES OPERATIONS

In principle, we followed the previous PN implementation described in Section 3.2. However, several important improvements were possible and/or necessary in order to handle very large text collections and the specific operations needed to translate XPath.

5.1 Index Structure

The index stores the initial and final positions of all the segments corresponding to nodes in the XML collection. These positions are stored as byte-offsets. Although a consecutive node or word numbering would suffice and yield smaller numbers needing less space, we chose byte-offsets in order to simplify the presentation of results to the user: given a node to display we know exactly which address of which file to access.

The index handles collections with multiple files. These are seen logically as a single large collection, where the content of each file is enclosed into `-DOCUMENT` tags. A small directory permits mapping virtual positions into the physical position of the proper file.

5.1.1 *Text Matching Index.* This is little more than an inverted index in secondary memory, where the set of all different words of the collection are maintained, and for each such word the list of all its occurrences are stored.

In order to efficiently solve phrase queries, the word-offsets of the words should be stored, as byte-offsets are not enough to distinguish whether two word positions form a phrase or not, especially because, in an XML context, there could be a lot of markup in the physical file between two words that appear as forming a phrase to an end-user.

On the other hand, we do not need to store byte-offsets of words. Byte-offsets, as explained, are necessary only to display the results. However, XPath does not permit to write queries that return simple words or phrases. Every answer must be an XML node. This reduces space requirements a lot.

Also, not all the text words have to be indexed. We manage a short list of words that will not be indexed (usually articles, prepositions, and other words that do not carry meaning). These are called *stopwords* and it is customary to remove them from indexes and queries in Information Retrieval scenarios [Baeza-Yates and Ribeiro-Neto 1999]. This permits saving up to 50% of index space at very little cost. In any case the list is configurable and the index can work under either decision.

In order to save index space, lists of consecutive positions are stored in a differential format: each number indicates the offset with respect to the previous. This poses no problems because all the lists are processed sequentially, and it yields smaller numbers. We take advantage of this by coding the offsets in an 8-1 format: the number uses as many bytes as necessary and the last bit of each byte is used to signal the end of the number. This coding is a good compromise between compression ratio and efficient handling.

As explained, words in attribute values have added a blank before their first position, and indexed as normal words. This makes it impossible to have text inside attributes as answers of non-attribute queries. Moreover, their word positions are accumulated in a separate counter, so that the presence of words in attributes does not disturb the result of a **same** operation regarding the text inside the node containing the attribute.

5.1.2 *Node Index.* Among the alternatives analyzed in the original implementation [Navarro 1995] we opted for the one that maintains a separate index for each different tag name. If we consider only the nodes with a given name, the result has also a tree structure (e.g. `section` in our small running example, see Figure 2 and also Figure 8). Hence each index stores a tree.

The tree is stored as a sequence of nodes, in depth-first order (a node, then recursively its children, then recursively its next sibling). There is no need to store a pointer to the first child of a node because, if it exists, it is right next to the node. A forward pointer to the next sibling is stored, and it points right next to the node if and only if the current node has no children. An additional advantage of this organization is that if more text is added at the end of the collection, we only need to append more nodes at the end of the index files, without need to rewrite them.

This organization permits answering queries consisting of tag names with a single pass over a contiguous file. All the algorithms ensure that (node or subexpression) trees are traversed using only two operations: *first-child* and *next-sibling*. These are extremely easy to execute in our format and ensure that we always move forward over the index files.

The decision of storing separate indexes per tag name favors tag-name queries against **Node** or **Attribute** queries, which can only be solved by a **union** of all the involved tag names. In practice these latter queries are very infrequent and most should be removed by algebraic optimization.

At index construction time, each node is labeled with a unique identifier. This is useful to know whether any two nodes are the same or not, and whether two nodes are children of the same parent. However, we do not need any additional storage for the node identifier: the byte-offset of its initial segment position is already unique, and we use it as the node identifier.

Since the node name is implicit from the index file the node is stored at, we only need to store, for each node, 6 numbers:

—The identifier (start tag position) of its parent in the whole hierarchy. This is not the same parent in the tree of the current index file, and it is essential for answering **parent/child** queries).

—Initial byte position and byte length of its text segment.

—Distance to its next sibling in the current index file. Actually, given that we have fixed node sizes, what we store is the subtree size, measured in number of nodes.

—Word-offset of first word, and number of words inside this node.

Word offsets are necessary in order to properly solve **same** queries: With byte offset information it is not possible to determine whether a node contains exactly a given sequence of words. Two reasons are the presence of attributes at the beginning of the segment covered by a node, and separator characters like whitespace at the extremes of the node. The first word of a node should be that of the next content word following the opening tag, so attributes are excluded. This is automatically obtained by keeping separate word-offsets for attribute values and other words. Figure 11 exemplifies the layout on disk.

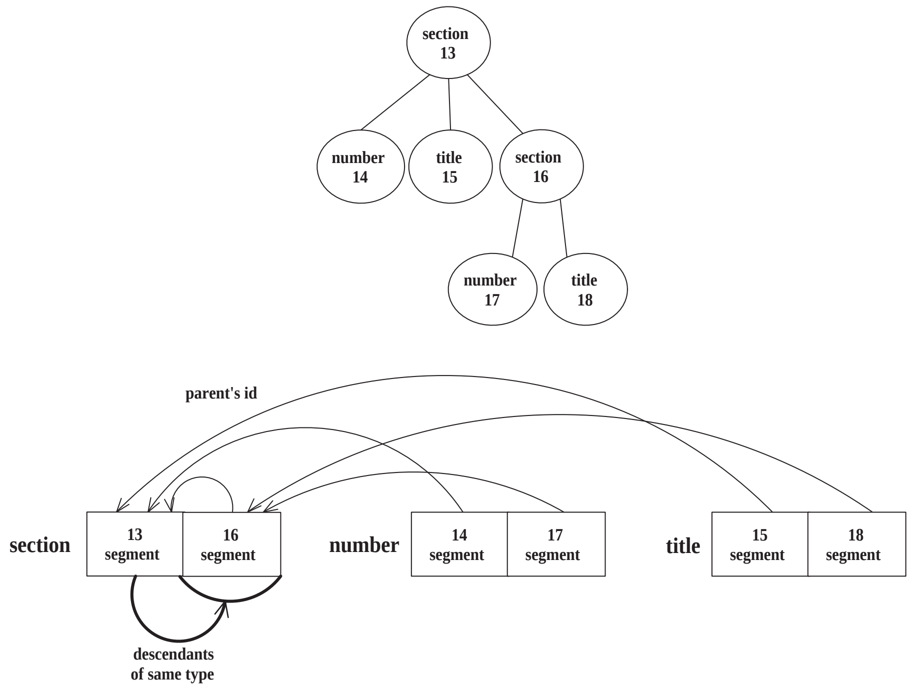


Fig. 11. Index file layout of a subtree of our running example. The numbers are the node identifiers, and segments are not detailed. The "parent's id" arrows are a graphical view of the value stored.

Some fields, such as initial text segment position and word-offset of the first word, could be compressed using 8-1 coding using differential encoding. However, we have to be careful because it is possible to arrive at a node from its parent, its previous sibling, or from descendants of the previous sibling. Other usually values, such as text segment length and word-offset of last word in the segment, cannot.

The reason is that we determine their values only after processing the last element of the node. By that time we have already written on disk the node data, and have to come back and write down these values, so we need to use a fixed amount of bytes. This is a consequence of our decision of storing nodes in preorder and of a one-pass construction.

Hence we need 6 numbers. In the databases we have examined, all these numbers are large enough to require full 4-byte integers. The only exception is the distance to next sibling, which is rather small (recall Table 4) and we encode it using 2-byte short integers. Hence, we need 22 bytes per node.

A structure-id is associated to each tag name at indexing time. This simplifies comparing node names.

At query time, it will be necessary to bring some nodes into main memory. Their amount is very low: in most cases, just one per query syntax tree element. In main memory, we need to associate some extra data to each node:

—Its structure-id (inherited from the index file the node was read from);

—its position in its index file (so that its children or siblings can be found if necessary); and

—document identifier to which the node belongs (necessary for **before** and **after** queries).

5.2 Lazy Evaluation

Two alternative evaluation schemes are proposed in the original PN implementation (Section 3.2). Since our focus is on large text databases, we cannot afford storing all the result of PN subexpressions in main memory before using them to compute other operations over these. Writing intermediate results to disk is also slow and cumbersome. Hence we chose lazy evaluation. However, the original work is not fully lazy: all the children of a given node are produced as soon as anyone is needed. Our current scheme is even more lazy.

We envision lazy evaluation as a process where we never build explicitly the results of PN expressions. Rather, we provide the mechanisms to *navigate* through the result trees. The navigation operations permitted are the same as for the index files: *first-child* and *next-sibling*. Hence, rather than implementing procedures that, given two result trees of subexpressions, compute a new result tree, we implement *cursors* that, given an operation and two cursors (that traverse subexpression results), are able to navigate through the result they should produce. Therefore, the results (final and intermediate) are never produced. Rather, we need to keep in main memory just one node of the result tree for each PN subexpression (the current node). This scheme works precisely because of the philosophy of the PN model: we can compute the result by traversing the arguments more or less in synchronization. The final result of the PN expression can be obtained incrementally, by navigating it with the *first-child* operation.

Actually, the operations have a slightly special semantics, as follows:

***first-child*:** moves to the first child of the current node. If it does not exist it moves to its next sibling. If no next sibling exists, it moves to the sibling of its parent, or of its grandparent, and so on.

***next-sibling*:** moves to the next sibling of the current node. If it does not exist it moves to the sibling of its parent, or of its grandparent, and so on.

Figure 12 shows an example. Actually, the query syntax tree is an active device throughout the query process. Each node is replaced by a cursor able to navigate through the result tree of the subexpression. We navigate the root node and show the final result tree. The navigation over the root node triggers navigation operations over subexpression nodes.

In particular, the cursors corresponding to tag-name queries are extremely simple. When the cursor is initialized the appropriate index file is opened. Each time we request the cursor to move to its *first-child*, it advances in the file by one position and delivers the current node. Each time we request the cursor to move to its *next-sibling*, it advances the file by its *number-of-descendants* field plus 1, and delivers the current node. Of course, buffering is used to reduce the amount of disk accesses. The scheme is very efficient and one can determine exactly how much main memory is going to be spent on buffering.

A similar scheme solves word matching queries. To initialize the cursor we search for the word in the vocabulary and fetch the list of its text occurrences. Both traversal operations are identical in this case: the next word position has to be delivered. Again we can use buffering to reduce disk accesses and at the same time use as much main memory as we want.

Hence the cursors for the leaves of the query syntax tree are easily implemented. The operations **Node** and **Attribute** are rewritten as a balanced **union** of all the known tag names of the appropriate type. In our example,

$$\begin{aligned}\textbf{Node} &= (((\texttt{chapter}\ \textbf{union}\ \texttt{section})\ \textbf{union}\ (\texttt{title}\ \textbf{union}\ \texttt{author}))\\ &\quad\ \textbf{union}\ (\texttt{image}\ \textbf{union}\ \texttt{book}))\\ \textbf{Attribute} &= ((\texttt{@caption}\ \textbf{union}\ \texttt{source})\ \textbf{union}\ \texttt{@number})\end{aligned}$$

where the balanced union ensures that each node traverses the hierarchy in time logarithmic in the number of different tags.

For the internal nodes, we need to implement a different procedure for each PN operation defined. This procedure is slightly different depending on whether we want the *first-child* or the *next-sibling*. At the invocation, all we know is the current node of the operands and the previous node delivered.

There is little point in going over all the 13 operations implemented plus their 13 negated versions. Rather, we prefer to show a few representative cases.

5.2.1 *Some Easy Operators.* In the seudocodes that follow, operations receive two subquery parameters P and Q, as well as a direction dir that can have the value $child$ or $sibling$, depending on where we have to move. Subqueries are manipulated as cursors, as explained. Field $X.result$ is the current value of cursor X (i.e., current node in the results of subquery X). Operation **Next(X,dir)** moves current cursor X according to dir, and returns the modified cursor X. Depending on the operator at the root of subquery X, **Next** becomes the appropriate PN operation (e.g. **In**). As the result of such a function invocation, we assign a new value to variable $result$, which becomes the current node of the corresponding cursor. Observe that, except for $union$, we always return values from P that satisfy some condition.

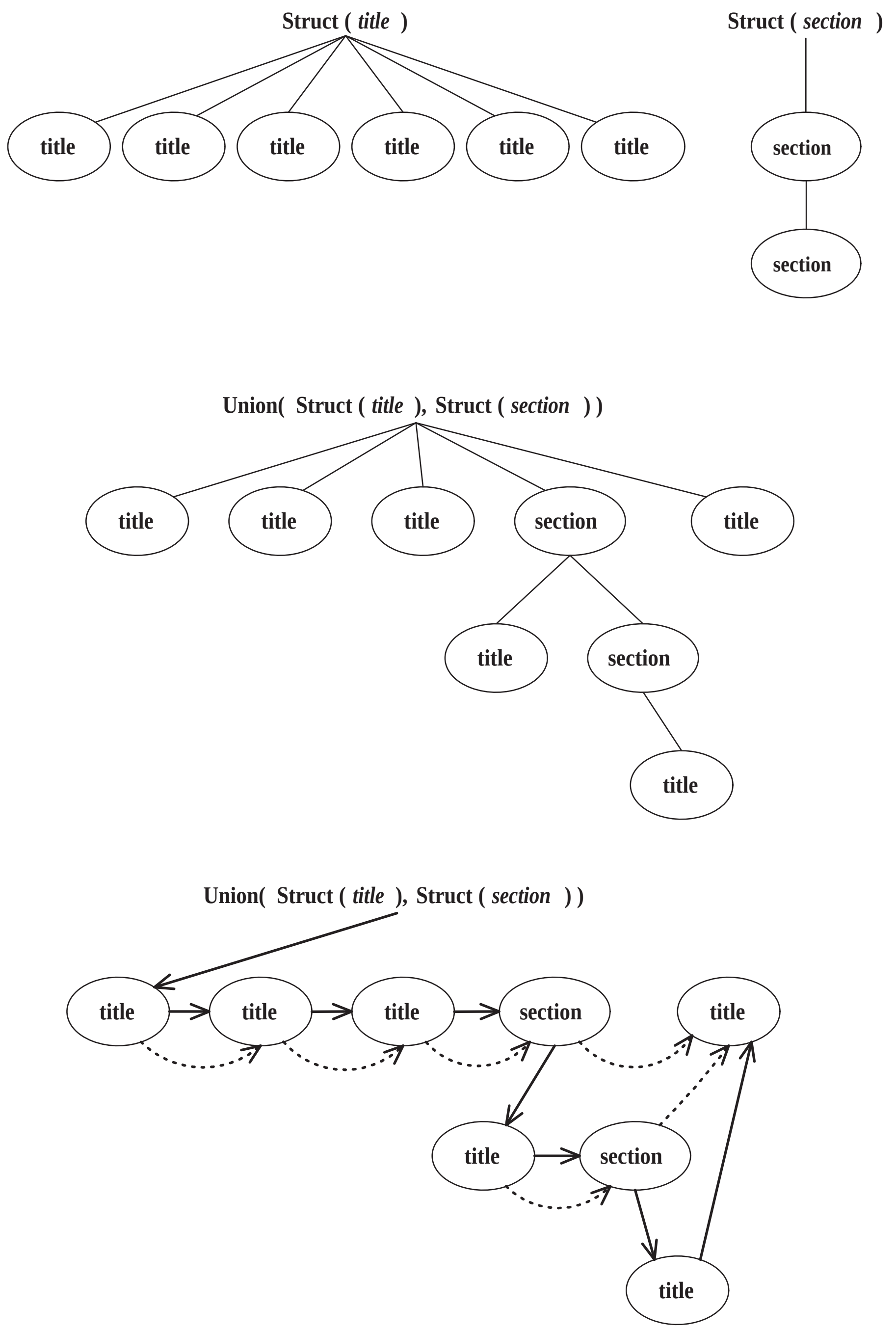


Fig. 12. Cursors over result trees. The top trees show the argument trees (obtained in lazy for from the index files). The middle tree is the whole result of the query `title` **union** `section`, but this tree is never produced. What one really has is a device like that of the bottom figure, where one can navigate using the *first-child* (solid line) and *next-sibling* (dashed line) operations.

A result equal to θ indicates that the tree traversal has finished. We can compare the segments of two nodes p and q by using the following relations: $p < q$ (segment of p is completely before segment of q), $p > q$ (idem after), $p \subset q$ (segment of p is strictly contained in segment of q), $p \supset q$ (strictly containing), and $p = q$ (segments coincide). Symbols $\leq$, $\geq$, $\subseteq$ and $\supseteq$ permit also the condition $p = q$ to be true.

A very simple example is the **In** function, which implements P **in** Q. As long as the current segments of P and Q are disjoint, it advances by *sibling* the leftmost segment. At some moment it finds a pair of nodes contained one in the other. If the node of P is not contained in that of Q, then it moves to the child of P, as some descendant of P could be contained in the current Q node. If, instead, the node of P is contained in the current node of Q, it stops at the current P node and this is the new *result*. In the beginning, it starts by moving in P by *dir*, since the invariant is that the current node in P has already been delivered.

Figure 5.2.1 gives the seudocode. If we replace $\supseteq$ by $\supset$ and $\subset$ by $\subseteq$ we obtain **Inself**. It should be clear that the time to traverse the result of "P **in** Q" is $O(|P| + |Q|)$, that is, linear in the size of the arguments, since we work $O(1)$ time per node of P or Q.

```
In (P, Q, dir)
1.    p ← Next(P, dir).result
2.    q ← Q.result
3.    While p ≠ θ AND q ≠ θ Do
4.        Case
5.            p < q : p ← Next(P, sibling).result
6.            p > q : q ← Next(Q, sibling).result
7.            p ⊇ q : p ← Next(P, child).result
8.            p ⊂ q : result ← p; Return
9.    result ← θ
```

Fig. 13. Operation **In**.

Let us now consider function **With**. It works quite similarly as **In**. This time we move to the child of Q if P is contained in the current node of Q, since there could be descendants of Q contained in the current P node. Another difference is that it is possible that Q is the result of a phrase query and hence it may represent a segment that overlaps structural segments. Hence we have used explicitly the *From* and *To* values of segments in order to move. Figure 5.2.1 gives the seudocode. Again, the same change as before yields **Withself**. The complexity is clearly linear as well.

Let us now consider function **Before**. It uses functor *Doc* over current node values, which is their document identifier (recall that this is stored when the node is in main memory). We advance in P or Q until they are in the same document. Then the result is the current P node if it is before the current q value. If current P node is after current Q node, we advance in Q by *sibling*. If one node includes

```
With $(P, Q, dir)$
1.  $p \leftarrow$ Next$(P, dir).result$
2.  $q \leftarrow Q.result$
3.  While $p \neq \theta$ AND $q \neq \theta$ Do
4.      Case
5.          $p \subseteq q : q \leftarrow Next(Q, child).result$
6.          $p \supset q : result \leftarrow p$; Return
7.          $From(p) < From(q) : p \leftarrow Next(P, sibling).result$
8.          $From(p) > From(q) : q \leftarrow Next(Q, sibling).result$
9.  $result \leftarrow \theta$
```

Fig. 14. Operation **With**.

another we advance in Q by *child*, as we cannot discard current P node until we are sure there is no relevant Q node ahead.

Figure 5.2.1 shows the seudocode, which is again clearly linear time. Function **After** is symmetric.

```
Before $(P, Q, dir)$
1.  $p \leftarrow$ Next$(P, dir).result$
2.  $q \leftarrow Q.result$
3.  While $p \neq \theta$ AND $q \neq \theta$ Do
4.      Case
5.          $Doc(p) > Doc(q) : q \leftarrow Next(Q, sibling).result$
6.          $Doc(p) < Doc(q) : p \leftarrow Next(P, sibling).result$
7.          $p > q : q \leftarrow Next(Q, sibling).result$
8.          $p < q : result \leftarrow p$; Return
9.          $else : q \leftarrow Next(Q, child).result$
10. $result \leftarrow \theta$
```

Fig. 15. Operation **Before**.

Other simple functions are those that implement **same** and **is**, as well as the negated versions of all the simple functions. The other operator that qualifies as "simple" is **union**. Although it needs some care because it is the only one where nodes from both P and Q can be retrieved, it is essentially simple. Finally, there are different versions for **same** and **with** when the right-hand operator is a text matching query, since in that case word-offsets rather than byte-offsets have to be used.

All simple operations need constant memory and linear time in the worst case.

5.2.2 *More Complicated Operators.* There are four PN operations that present complications. Two of them, **child** and **after-sibling**, still achieve linear worst-case time but need $O(h)$ space, being h the height of the collection tree. This is not really a problem in practice, as h is usually very low. The other two, **parent** and **before-sibling**, are worse. We obtain linear time only on average ($O(n \log n)$ worst case), and need potentially $O(n)$ space.

We note that the key issue is that these are the operations that make direct reference to the structure of the whole collection: their results cannot be determined by looking at segment inclusion only, but we need to consider direct parentship in the XML tree. On top of that, the operations corresponding to forward axes only require to keep, for the current node, the list of its ancestors in the result tree, which are only $O(h)$ and have already been computed. The reverse axes, on the other hand, require all their descendants, which are $O(n)$ and have to be computead ahead of time.

Let us first consider "P **child** Q". The problem is that, given a current P and Q nodes, such that $P \subset Q$, we may have to enter inside Q in order to find the parents of some other nodes inside P, but later it may be that the current value of Q is the correct parent of a subsequent node of P. Figure 16 illustrates this case. If we start at P_1 and Q_1, we must move to Q_2 in order to properly find the parent of P_2, but later, when we move to P_3, we should come back to Q_1 to find the parent of P_3.

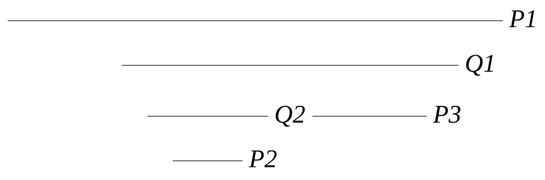


Fig. 16. A case where we cannot advance in Q and forget the ancestors.

Moving backwards goes against all the philosophy of the model. So we prefer to store a *stack* of ancestors of the current Q node. These ancestors refer to the virtual tree Q and have been already seen. We also keep the invariant that $p \subseteq Top(stack)$.

Figure 5.2.2 gives the seudocode. We use functor Id for nodes, which as explained is just the byte offset of the segment beginning. We also use $Parent$, which is the parent identifier, i.e., parent byte offset again.

Although the stack is $O(h)$ space, the algorithm is still linear time in the worst case. This is not so immediate this time: we can perform several *Pop* operations for a single Q node. However, there cannot be more *Pop*'s than $Push$'s overall, and these are linear overall.

The algorithm for **after-sibling** suffers from the same problem: Siblings of any ancestor of the current P node can appear later, after we have processed other descendants. The solution uses a similar stack, and the space and time complexity stays the same.

Let us consider now the reverse axes with the same problem, in particular the *parent* operation. Given current nodes of P and Q, we may need to traverse all the

```
Child ($P$, $Q$, $dir$)
1.  $p \leftarrow$ Next$(P, dir).result$
2.  $q \leftarrow Q.result$
3.  While $p \neq \theta$ AND $q \neq \theta$ Do
4.      While $p \not\subset Top(stack)$ Do $Pop(stack)$
5.      Case
6.          $Parent(p) = Id(Top(stack))$ : $result \leftarrow p$; Return
7.          $p < q$ : $p \leftarrow Next(P, sibling).result$
8.          $p > q$ : $q \leftarrow Next(Q, sibling).result$
9.                  While $q \not\subset Top(stack)$ Do $Pop(stack)$
10.         $p \supseteq q$ : $p \leftarrow Next(P, child).result$
11.         $p \subset q$ : $Push(stack, q)$;
12.                 $q \leftarrow Next(Q, child).result$
13.                 While $q \not\subset Top(stack)$ Do $Pop(stack)$
14. While $p \neq \theta$ AND NOT $IsEmpty(stack)$ Do
15.     Case
16.         $Parent(p) = Id(Top(stack))$ : $result \leftarrow p$; Return
17.         $else$ : $Pop(stack)$
18. $result \leftarrow \theta$
```

Fig. 17. Operation **Child**.

Q nodes that descend from the current P node before finding a child that selects the current P node. The Q nodes traversed before that must be remembered, however, because they may be necessary to select further nodes of P. Figure 16 serves again as an example if we consider "Q **parent** P". If the current nodes are Q_1 and P_2, we need to traverse P_2 and P_3 in order to know that Q_1 qualifies, but then we have forgotten P_2, which is necessary to make Q_2 qualify.

The solution is to store the Q descendants of the current P node in a hash table, indexed by their *parent-id* value. Hence the identifier of subsequent P nodes are searched for in this table. Figure 5.2.2 gives the seudocode.

Note that we only need to know whether the child of a given P node has been inserted in the hash table, so a bit array suffices, and this is the way it is implemented in the prototype. In case we prefer to use a classical table with only the relevant P nodes inserted, we could implement a mechanism to remove old P nodes once we have definitely abandoned their area. This can be implemented as part of the same $Insert$ function: when inserting a new node p, every other node p' such that $To(p') < From(p)$ can be removed from the hash table.

This algorithm is linear time on average, and it could be made $O(n \log n)$ in the worst case (where $n = |P| + |Q|$), by using a balanced binary search tree. It requires in the worst case enough space to store the whole argument P. This could even be stored on disk, which would slow down the process but permit an implementation with bounded main memory. Our prototype stores the hash table in main memory.

The situation with **before-sibling** is similar. This time, however, the hash table

```
**Parent** $(P, Q, dir)$
1.    $p \leftarrow \textbf{Next}(P, dir).result$
2.    $q \leftarrow Q.result$
3.    **While** $p \neq \theta$ AND $q \neq \theta$ **Do**
4.        $Insert(hash, Parent(q))$
5.        **Case**
6.            $Exists(hash, Id(p)) : result \leftarrow p$; **Return**
7.            $p < q : p \leftarrow Next(P, sibling).result$
8.            $p > q : q \leftarrow Next(Q, sibling).result$
9.            $else : q \leftarrow Next(Q, child).result$
10.   **While** $p \neq \theta$ **Do**
11.       **Case**
12.           $Exists(hash, Id(p)) : result \leftarrow p$; **Return**
13.           $else : p \leftarrow Next(P, sibling).result$
14.   $result \leftarrow \theta$
```

Fig. 18. Operation **Parent**.

is a bit bigger because we need also to store, for each parent identifier in the hash table, which is the last sibling that qualified, so as to know whether the current P node is before that Q sibling. We use a classical closed hashing table, without removing obsolete nodes.

6. A SOFTWARE PROTOTYPE

We have implemented a software prototype called IXPN (Index for XPath using Proximal Nodes). An online demo can be seen in `www.dcc.uchile.cl/ixpn`. The demo indexes several of the databases described in the experiments and permits executing XPath queries against them. The translation to PN is shown both in plain format and as a query syntax tree. The result of the query can also be examined.

In this section we describe the prototype and our experimental performance comparisons against other existing softwares to solve XPath queries.

6.1 Description

The software prototype consists of three components (see Figure 19):

—An indexer, which builds the text and structure indexes from the XML collection;

—a query evaluator, which receives and XPath expression and returns the qualifying nodes; and

—a visualizer of results, which shows the XML content of the resulting nodes.

All the software was developed in C language, using a function-oriented modular scheme.

6.1.1 *Indexer.* To build the document indexer we used a fast and flexible XML parser called *Expat* [Clark and Cooper 2002].

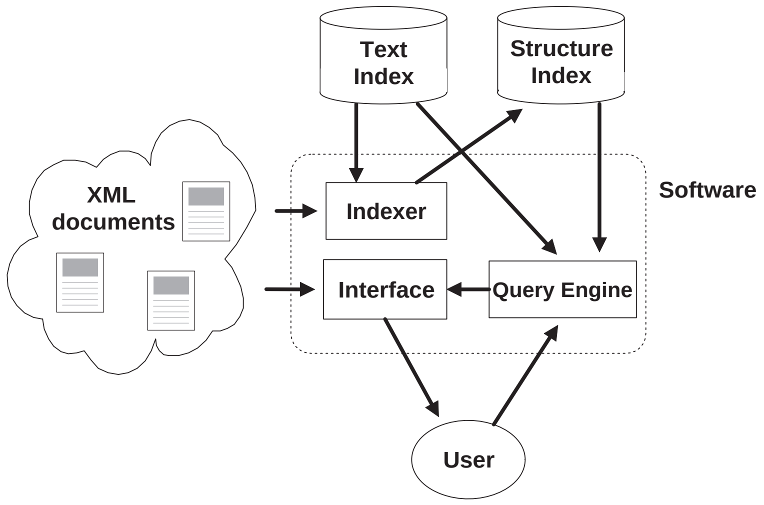


Fig. 19. The architecture of IXPN.

As it reads the documents of the collection, the indexer writes to disk the structure nodes and builds in main memory a trie data structure with the text words and their positions. When the memory used by the trie reaches a given limit, a *partial index* is stored to disk. Finally, partial indexes are *merged* in a balanced way, and other information such as document list, tag list, etc. is generated. This follows the general scheme to build an inverted file depicted in [Baeza-Yates and Ribeiro-Neto 1999].

About 70% of the index time is used to store and combine the partial indexes on disk.

The total main memory required by the indexer is determined by the height of the XML tree. This is because a node can be stored on disk only when its final position is known, so we may have to keep in main memory a whole path of nodes before writing them do disk. This requirement is in practice minimal. A more relevant requirement is that of the trie data structure, but this can be fixed almost arbitrarily and traded for indexing time. This means that the amount of main memory available is usually not an issue.

6.1.2 *Query Evaluator.* We used *flex* [GNU Project 2000] and *bison* [GNU Project 2003] for the syntax analysis of the XPath query. The evaluation is done in three steps:

(1) Construction of the PN query syntax tree.
(2) Modification of the syntax tree to account for simplifications and other transformations.
(3) Evaluation of the PN query. A loop sequentially obtains the results, under the lazy evaluation scheme.

6.1.3 *Visualization of Results.* For this prototype we chose to present to the user the original XML pieces corresponding to each returned node. This component could be easily replaced by others that execute more sophisticated visualization or even pass the results in some predefined format to other higher level query processors, such as an XQuery processor.

6.2 Experimental Performace

We measured the performance of IXPN in evaluating different kinds of simple and complex queries.

6.2.1 *Setup.* We used a dedicated 700 MHz Intel Pentium III with 384 Mb of RAM running Windows XP 2002, with a 30 Gb local hard disk Maxtor 5400 RPM. Each experiment was repeated 20 times and the average elapsed times are reported, in milliseconds (msecs). Standard deviation was 5 to 10 msecs. Given our lazy evaluation scheme, we measured the time to retrieve 1, 10, 100, and all the answers of each query. The time to obtain the first node is taken as the *latency* of the query time (open files, fill buffers, etc.), and then a *time per node* is computed by subtracting the latency and dividing the remaining time by the number of nodes returned by the query, considering the time to retrieve all the answers.

We counted the times to obtain the node identifiers, not that of outputting the text content of each node, as this is a feature external to the engine.

6.2.2 *Text Collections.* We used XML collections from four different sources and with different characteristics:

Shakespeare: A collection of plays from Shakespeare [Bosak 1999b].

GCIDE: A collaborative dictionary, compiled the GNU Project [Dyck 2002].

Religion: A collection of religious texts [Bosak 1999a].

DOE: Short abstracts from DOE publications [Harman 1995].

These collections have very diverse level of structuring. We measure it in terms of the percentage of the total collection size that is used by XML tags. This factor strongly influences the space required by the structure index. Table 3 gives several relevant parameters, including space overhead of both indexes. We use a set of 122 stopwords, formed by prepositions, articles, and so on. This produced a 50% reduction in the space for the inverted index.

Collection	Size (Mb)	# docs.	# tags	# attr.	% struct.	txt-idx	str-idx
Shakespeare	10.0	37	21	1	50.49%	45.0%	75.0%
GCIDE	53.5	28	289	6	39.53%	17.3%	83.1%
Religion	6.7	4	28	0	5.50%	15.0%	16.5%
DOE	91.5	93	4	1	4.55%	19.8%	7.6%

Table 3. Some data on the XML collections used. By "# tags" and "# attr." we refer to the number of different tag and attribute names, respectively. "% struct" refers to the level of structuring. The last two columns show the space overhead of text and structure indexes.

The size of IXPN indexes changes drastically depending on the structuring level of the collection, since each node occupies a fixed amount of space on disk, usually much larger than the text length of the corresponding tag. Collections Shakespeare and GCIDE have a high level of structuring as compared to Religion and DOE. This accounts for the incidence of the two indexes on the overall space overhead.

It might also be interesting to see which are the maximum values of the fields stored at index nodes, so as to evaluate the possibility of compression. Table 4 shows this. As it can be seen, most numbers are rather large, so compression is not trivial. Of course, it would be possible to consider the maxima of each tag name separately in order to compress those with smaller fields.

The main surprise might be that the distance to the sibling is always one node. This is because, in all our example databases (those shown here and others omitted), a tag cannot contain another tag of the same name. We will consider some consequences of this in the conclusions.

Collection	Parent-id	First-byte	Byte-len	Dist-sibl	First-word	Word-len
SHAKESPEARE	10,479,622	10,180,162	10,479,683	22	648,946	23,479
GCIDE	56,109,897	55,975,405	56,078,405	22	3,571,749	399,844
RELIGION	6,997,786	3,511,727	6,998,843	22	550,181	287,987
DOE	96,030,726	95,340,608	96,031,587	22	9,043,007	105,532

Table 4. Maximum size of different fields for the XML collections used.

6.2.3 *Queries.* We tested each operation in isolation in order to analyze the performance of the different functions implemented. Later we show tests on complex queries. The queries have to be different for each collection because they have different tags. However, we use a general scheme and change only tag names. These are:

Child: Queries of the form `struct1/struct2`.

Parent: Queries of the form `struct1[struct2]`.

In: Queries of the form `struct1//struct2`.

With: Queries of the form `struct1[//struct2]`.

Following: Queries of the form `struct1/following::struct2`.

Preceding: Queries of the form `struct1/preceding::struct2`.

Following-sibling: Queries of the form `struct1/following-sibling::struct2`. This is abbreviated in the tables as `struct1/foll-sibling::struct2`.

Preceding-sibling: Queries of the form `struct1/preceding-sibling::struct2`. This is abbreviated in the tables as `struct1/prec-sibling::struct2`.

Text: Queries of the form `struct[.=~"word"]`.

Phrase: Queries of the form `struct1[.=~"word1 word2"]`.

Node: Queries of the form `*`.

Attribute: Queries of the form `@*`.

Note that "`*`" and "`@*`" are in fact tests for the speed of **union**, as the query is translated into a balanced union of all the tag names. This has to be taken into account when the number of "nodes involved" is computed, as we refer to all intermediate results. Not only the original arguments are counted, but also the internal nodes of the query syntax tree. This includes the final result, corresponding to the root of the syntax tree.

6.2.4 *Results.* Tables 5, 6, 7 and 8 show the results. As it can be seen, the latency is rather constant, from 70 to 90 msecs in most cases. An exception is for the query "*" on GCIDE, due to the large number of different tags, and hence of leaves in the query syntax tree. An initial buffer of results has to be filled for each such leaf.

It is also clear that, once this latency is paid, the time to retrieve 1 or 100 nodes is not very different. On the other hand, the type of operation and collection type or size do not have much influence.

In general, latency excluded, IXPN takes 15 to 35 microseconds (μsecs) to output each new answer node. Obtaining a better approximation is difficult because the time depends not only on the size of the result but also on the sizes of the intermediate results (called "nodes involved" in the tables). It can be seen that there are a few cases where the time per node is much larger than 35 μsecs. In most of these cases, the number of nodes involved exceed by a factor of 10 the answer size. This is usually the case of **parent**, **with**, and **phrase** operations. Trying to model the time as a function of nodes involved does not help, because these operators usually skip a large amount of involved nodes, so they take much less time per involved node than others. The remaining cases of very large time per node answered corresponds to queries that anyway return too few answers.

It is also interesting that the operations that were algorithmically problematic have worked well in practice, for example **child** and **parent**. They are as fast as the simpler **in** and **with**.

Query	Nodes retrieved				Answer	Nodes	μsecs
	1	10	100	All	size	involved	/node
`SPEECH/LINE`	80	80	81	2,060	107,833	246,694	18
`SPEECH[LINE]`	80	80	84	913	31,028	169,889	27
`SCENE//LINE`	97	97	98	1,915	107,164	215,747	17
`SCENE[//LINE]`	101	103	114	211	750	109,333	147
`LINE/following::LINE`	87	87	89	2,667	107,796	323,462	24
`LINE/preceding::LINE`	85	85	88	2,788	107,796	323,462	25
`LINE/foll-sibling::LINE`	85	85	89	3,398	76,805	292,471	43
`LINE/prec-sibling::LINE`	83	84	89	2,126	76,805	292,471	27
`LINE[.=~"love"]`	86	86	94	209	1,705	112,600	72
`SPEAKER[.=~"MARK ANTONY"]`	87	87	94	102	204	31,739	74
`*`	110	111	112	5,778	179,689	1,018,237	32
`@*`	88	88	97	3,742	179,689	179,689	20

Table 5. Elapsed time to solve different queries on the SHAKESPEARE collection. Times are in msecs. The time for 1 node is the latency and "μsecs/node" refers to microseconds per answer node, latency excluded.

6.3 Comparison against Others

Although there exist many prototypes and test versions of softwares that support XML databases, most of them are commercial developments. In the best cases, online demos are available via Web, but these cannot be used for comparison purposes because of different server architectures, different text collections, and even because of the network latencies that distort the results.

Query	Nodes retrieved				Answer size	Nodes involved	μsecs /node
	1	10	100	All			
`p/source`	84	85	88	4,548	229,043	689,674	19
`p[source]`	84	84	86	1,041	8,568	469,199	112
`p//br`	83	83	84	4,273	243,885	718,786	17
`p[//br]`	85	86	88	4,347	226,203	701,104	19
`p/following::p`	90	93	92	5,651	230,989	693,021	24
`p/preceding::p`	80	83	88	5,850	230,989	693,021	25
`p/foll-sibling::p`	87	88	96	6,729	230,989	693,021	29
`p/prec-sibling::p`	84	85	89	5,892	230,989	693,021	25
`p[.=~"Webster"]`	73	75	76	1,262	25,722	469,160	46
`p[.=~"1913 Webster"]`	71	72	77	1,510	24,873	680,733	58
`*`	3,086	3,086	3,087	126,943	2,201,761	21,469,074	56
`@*`	92	98	—	100	60	220	133

Table 6. Elapsed time to solve different queries on the GCIDE collection. Times are in msecs. The time for 1 node is the latency and "μsecs/node" refers to microseconds per answer node, latency excluded.

Query	Nodes retrieved				Answer size	Nodes involved	μsecs /node
	1	10	100	All			
`book/chapter`	82	82	85	104	1,423	2,922	15
`chapter[v]`	102	102	110	250	1,090	46,462	136
`tstmt//v`	84	84	86	737	43,949	87,902	15
`chapter[//v]`	87	88	93	149	1,423	46,795	44
`title/following::v`	82	83	85	996	43,949	87,906	21
`v/preceding::v`	82	82	83	1,080	43,945	131,843	23
`v/foll-sibling::v`	80	81	85	1,172	42,386	130,284	26
`title/prec-sibling::v`	82	82	91	1,166	42,386	86,343	26
`v[.=~"God"]`	88	89	96	202	4,404	53,383	26
`v[.=~"LORD God"]`	92	93	101	212	1,113	55,710	108
`*`	111	113	116	1,517	48,259	275,766	29
`@*`	—	—	—	—	—	—	—

Table 7. Elapsed time to solve different queries on the RELIGION collection. Times are in msecs. The time for 1 node is the latency and "μsecs/node" refers to microseconds per answer node, latency excluded.

Query	Nodes retrieved				Answer	Nodes	μsecs
	1	10	100	All	size	involved	/node
DOC/DOCNO	83	83	85	2,175	112,144	336,372	19
DOC[DOCNO]	80	80	83	2,279	112,144	336,372	20
DOC//TEXT	86	87	90	2,044	112,144	336,372	17
DOC[//TEXT]	86	86	89	2,201	112,144	336,372	19
DOC/following::DOC	88	89	93	3,041	112,054	336,282	26
DOC/preceding::DOC	80	81	85	2,787	112,054	336,282	24
DOC/foll-sibling::DOC	79	79	82	3,045	112,054	336,282	26
DOC/prec-sibling::DOC	82	83	89	2,830	112,054	336,282	25
TEXT[.=~"energy"]	70	70	73	765	19,102	162,060	36
TEXT[.=~"high energy"]	71	73	99	609	1,205	180,656	446
*	93	96	97	9,290	336,522	1,009,566	27
@*	89	90	—	91	90	90	22

Table 8. Elapsed time to solve different queries on the DOE collection. Times are in msecs. The time for 1 node is the latency and "μsecs/node" refers to microseconds per answer node, latency excluded.

We obtained six softwares whose source or executable versions were available, and compared them against IXPN. These are

Xindice [Apache Software Foundation 2002]: Indexes documents using a native XML database with proprietary format. It is designed to work on small and medium-size collections, with a maximum document size of about 5 Mb. It uses the technology of Apache group to work with XML documents, which consists of a set of Java classes . Queries are run on a server process. The indexes are stored in a compressed format and accessed from disk. Xindice implements only a basic XPath functionality.

eXist [Meier 2002]: Indexes documents using a native XML database with proprietary format. It is designed to work on small and medium-size collections, with a maximum document size of about 5 Mb. It uses the technology of Apache group to work with XML documents, which consists of a set of Java classes . Queries are run on a server process. Indexes are stored and managed on disk. eXist implements a complete XPath functionality.

XMLGrep [Jones 2000]: Searches the documents sequentially, looking for regular expressions the XPath queries are transformed into. It is implemented in C language and only supports basic XPath operations. This project was abandoned by its developer. It cannot handle multiple-document collections.

Saxon [Kay 2002]: Searches the documents sequentially, but it builds the structure tree of each document before running the query against it. It is implemented as Java classes. Saxon is oriented to transforming XML documents using XSLT language [Consortium 1999b], but it can be adapted to solve XPath queries. It implements lazy evaluation for XPath. It cannot handle multiple-document collections.

MSXML [Microsoft Corp. 2002]: Searches the documents sequentially, but it builds the structure tree of each document before running the query against it. It is an API available as a COM component for several Microsoft languages such as C++, VisualScript and JScript. MSXML is oriented to transforming XML documents using XSLT language [Consortium 1999b], but it can be adapted to solve XPath queries. It cannot handle multiple-document collections. MSXML is currently considered to be one of the most efficient developments in technologies for XML management.

ToXin [Toronto XML Server Project 2002]: Searches the documents sequentially, but it builds the structure tree of each document before running the query against it. It is implemented as Java classes. It implements a highly simplified version of XPath that includes only the axes `child` and `descendant`, which are called "regular expressions" of XPath.

6.3.1 *Indexing.* Table 9 compares the time and space necessary to index our test text collections. We only consider IXPN, Xindice and eXist, since the others do not build any index but sequentially scan the collection for every query. We let IXPN use 10 Mb of RAM to index the text.

Collection	IXPN			Xindice			eXist		
	Time	Speed	Size	Time	Speed	Size	Time	Speed	Size
SHAKESPEARE	34	0.294	120%	67	0.149	83%	337	0.030	410%
GCIDE	217	0.244	100%	437	0.122	74%	—	—	—
RELIGION	9	0.746	32%	23	0.294	133%	80	0.084	235%
DOE	222	0.416	28%	222	0.416	119%	—	—	—

Table 9. Time and space to index the test XML collections. Time is measured in seconds, speed in Mb/sec, and size in extra percentage over the XML text size.

Xindice compresses and stores the XML documents, unlike IXPN, which retains the original documents (hence in order to compare space overheads we should add 100% to IXPN). This is the reason why Xindice had more overhead on less structured collections. Moreover, IXPN does not index stopwords. This makes it difficult to compare the respective index sizes. However, it is interesting that both indexes have similar space overheads on little structured collections.

eXist, on the other hand, could not index the larger collections GCIDE and DOE, because of excessive memory requirements. The indexes produced are huge, although it answers queries faster than Xindice.

IXPN was the fastest to produce the index, at a rate of 1.3–4.0 secs/Mb. Next was Xindice, with 2.4–8.2 secs/Mb, and the slowest was eXist, at a rate of 12–33 secs/Mb.

6.3.2 *Searching.* We tested more complex queries against the collections RELIGION, SHAKESPEARE and DOE. All queries are evaluated from the root of the tree, as required by the other softwares (not IXPN). The query syntax was adapted to each software. Java, JScript and Perl programs were developed as necessary to test them, in particular for those unable to process several documents simultaneously. Since the other softwares return the text content of returned nodes, IXPN was modified to do the same. We measured the time to return all the results, using an external software for fairness.

Tables 10, 11 and 12 show the results. We use some obvious abbreviations for the software names. col chica los seq ok, grandes no. malos para //, toxin no daba con algunos

We note that the softwares developed in Java are much slower than the rest. The exception is ToXin. However, for this program we measured only the time to execute the query, disregarding the time to build the index in main memory. This can be fair if we measure performance in hot state, although for the others we measured time in cold state.

Xindice and eXist use too much main memory, close to 100 Mb. They were not able to build their in-memory indexes for GCIDE and DOE, and Xindice could not answer any query on SHAKESPEARE. Xindice is very slow in general, but especially with operator "//". This is mentioned in the documentation, where it is recommended to omit it close to the root of the collection. In fact, all the implementations recommend the same. The reason is that they operate by traversing the tree directed by the axes, and operator "//" forces them to traverse the whole tree. For the same reason, all them require the queries to start at the root of the tree. This is a clear advantage of IXPN, which works bottom-up and is very efficient for this type of operation.

XMLGrep performs bad on reverse axes, which require it to go back to check pieces of documents already traversed. Note also that only IXPN and ToXin are able of quickly determining that a given structure tag does not exist (last query on DOE).

Sequential search solutions, such XMLGrep, work well on small collections, but it is too slow on large sets. The same performance is exhibited by Saxon and eXist. MSXML and ToXin, on the other hand, handle large collections better. However, none of these can be considered a competitive choice for handling a large text collection (several hundred megabytes). In addition, ToXin handles a very limited subset of XPath, which excludes several of our example queries.

IXPN, on the other hand, performed well for small and large text collections, taking usually less than 2 seconds to answer queries. It was by far faster than all the other alternatives and does not seem to be much affected by the size of the collection.

Query	IXPN	Xind	eXist	Grep	Saxon	MS	ToXin
`/tstmt/bookcoll/book/chapter`	**1.8**	20.5	8.8	3.4	4.0	3.3	2.5
`/tstmt/coverpg/coverpg[title]`	**0.5**	2.8	2.2	0.7	3.3	1.3	—
`/tstmt//chapter`	**1.8**	58.9	8.8	3.8	4.1	3.2	2.5
`/tstmt[//chapter]`	**0.9**	22.7	8.8	3.7	4.0	4.2	—
`v[.=~"love"]`	**0.4**	9.9	9.8	0.7	3.4	1.8	3.7
`/tstmt/coverpg/title` `/following-sibling::subtitle`	**0.5**	2.6	9.8	0.7	3.3	1.3	—

Table 10. Elapsed time, in seconds, to solve different complex queries on the softwares tested over collection RELIGION.

Query	IXPN	Xind	eXist	Grep	Saxon	MS	ToXin
`/SPEECH[SPEAKER="mark antony"] /LINE`	**0.1**	—	25.6	24.5	23.2	5.5	—
`PLAY[TITLE=~"hamlet"] //PERSONA`	**0.1**	—	25.7	24.4	23.8	5.4	—
`SCENE[//SPEAKER="romeo" and //SPEAKER="juliet"] /TITLE`	**0.1**	—	12.2	38.1	24.2	8.1	—
`PLAY[//ACT/TITLE=~"act III"] /TITLE`	**0.6**	—	20.1	25.1	23.7	5.7	—
`SPEECH[SPEAKER="juliet"] /preceding-sibling::SPEECH [SPEAKER="romeo"] /ancestor::SCENE/TITLE`	**0.2**	—	12.4	23.5	23.1	5.8	—

Table 11. Elapsed time, in seconds, to solve different complex queries on the softwares tested over collection Shakespeare.

Query	IXPN	Xind	eXist	Grep	Saxon	MS	ToXin
`/FILE/DOC/DOCNO`	**2.3**	—	—	13.8	61.0	24.4	3.6
`/FILE//TEXT`	**1.9**	—	—	68.4	67.7	47.1	19.0
`/FILE/@*`	**0.1**	—	—	12.5	62.2	12.1	—
`/FILE/DOC/DOCNO[TEXT]`	**1.6**	—	—	130.2	61.3	11.5	—
`/FILE/DOC/TEXT[.=~"energy"]`	**0.8**	—	—	65.3	69.7	20.2	—
`/*`	**7.6**	—	—	58.9	69.3	60.0	22.5
`//AAA`	<**0.1**	—	—	13.6	60.0	11.4	0.2

Table 12. Elapsed time, in seconds, to solve different complex queries on the softwares tested over collection DOE.

7. CONCLUSIONS

We have presented IXPN, an indexed search technique to answer XPath queries over large XML collections. IXPN first builds an index on disk over the XML collection. Based on that index, it is able of answering XPath queries over the collection. IXPN works in a lazy manner, so the answer can be retrieved incrementally and navigated through, for example discarding uninteresting answer subtrees without need to even producing it. IXPN can index and query an arbitrarily large text collection with a very limited main memory; in most cases as limited as desired. The current prototype of IXPN can be tested at `http://www.dcc.uchile.cl/ixpn`.

We have focused on the "most interesting" part of XPath functionality, leaving aside the programming-language-like features (`http://www.w3.org/TR/xpath-#corelib`). These depend on the embedding language and are easier to implement efficiently. We also disregard instruction-nodes, namespaces, etc., whose inclusion is rather trivial. References are also not considered, but these are part of other languages that contain XPath, such as XLink.

IXPN is based on Proximal Nodes (PN), a generic model to query structured text. We have shown how, despite looking very different, XPath can be converted into PN. We have reimplemented the PN model in a more memory-efficient way, and at the same time have reduced disk overheads to a minimum. All the operations work in time linear on the size of the arguments (most in the worst case, a few on average). Most operations require constant space, although some require space proportional to the height of the XML tree, and a few pathological cases could require memory proportional to one of the arguments.

We have shown that IXPN is by far more efficient than all the publicly available alternatives we were aware of, including MSXML. In particular, IXPN was the only one unaffected by the collection size, and in fact the only one that can currently be seriously considered to handle large text collections. IXPN is also unaffected by the use of the '//' operator, which is troublesome for all other softwares. This is due to the bottom-up nature of PN algorithms, as other alternatives traverse the structure tree in a top-down fashion.

We are working on the current prototype in order to improve the compression of the index (which is currently very basic but already competitive), and on including more algebraic optimization of PN queries, which can make a large difference in ill-posed queries. In particular, we have observed that containment between nodes of the same type is very rare. Indeed, it is so rare that we could remove the pointer to the next sibling in our structure index, and in case we need to move to the sibling we could just move sequentially (as if we moved to children) until reaching the sibling. This would save 10% of structure index space and the effect on query time would be minimal.

Other important aspects not yet considered are: handling transactions, implementing an API to give access to IXPN via programming languages, multicollection support, handling updates to the text via efficient reindexing, developing a client-server architecture, clever handling of frequent queries, etc.

An issue that deserves more research is how to efficiently deal with direct edge queries (those involving child–parent and sibling relationships). These have been the only where we could not guarantee linear time and constant space. From these, reverse axes were the most complicated. Although we showed that in practice there is no big difference, there is an intrinsic problem related to the lazy evaluation of these operations, as it is not always possible to run them by moving forward. As we forced that, we had to precompute some results ahead of time and storing them for later.

On the other hand, these direct edges are easily dealt-with by the usual top-down approach, for which our easyness to handle transitive operations (descendant/ancestor, for example) is difficult to achieve. It would be interesting to join the best of both approaches.

We are also interested in extending our XPath implementation to include all the operators of the standard, as well as other operations not included but that we could handle efficiently, such as searching allowing errors, searching for regular expressions, and so on. More importantly, we plan to handle more sophisticated embedding languages, such as XQuery or XSLT, keeping the current efficiency as much as possible.

Finally, it should be pointed out that IXPN could be used as a sequential engine, to work on an XML stream without any index. A low-level scanner would traverse the text, recognizing words and structural nodes that are mentioned in the query and filling buffers of answers at the leaves. The rest could proceed in lazy form as more data becomes available at the leaves. We believe that this could be competitive against current sequential alternatives that either search for regular expressions or explicitly build the structure tree.

Another interesting idea is to rewrite a document collection as a sequence of node and word identifiers. This would yield a compressed representation of the collection, and with the aid of the index it would be possible to reproduce a rather legible version of the document. This is interesting, for example, in Web search engines that maintain a simple version of all the text contents. It might be interesting to focus on compressed indexes for XML collections, as done in [Ferragina and Mastroianni].

REFERENCES

Apache Software Foundation. 2002. XIndice. `http://xml.apache.org/xindice`.

Baeza-Yates, R. and Navarro, G. 2000. XQL and proximal nodes (preliminary version). In *Proc. XML Workshop of SIGIR'2000 (23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2000). Extended version to appear in *JASIST*.

Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval.* Addison-Wesley.

Bosak, J. 1999a. Religion 2.0. `http://www.ibiblio.org/bosak`.

Bosak, J. 1999b. Shakespeare in XML. `http://www.ibiblio.org/xml/examples/-shakespeare`.

Clark, J. and Cooper, C. 2002. The Expat XML parser. `http://expat.sourceforge.net`.

Consortium, W. 1999a. XPath 1.0: XML path language. Technical report, WWW Consortium. `www.w3.org/TR/xpath/`.

Consortium, W. 1999b. XSL transformations (XSLT). Technical report, WWW Consortium. `www.w3.org/TR/xslt/`.

Consortium, W. 2001a. XML linking language (XLink) version 1.0. Technical report, WWW Consortium. `www.w3.org/TR/xlink/`.

Consortium, W. 2001b. XML pointer language (XPointer) version 1.0. Technical report, WWW Consortium. `www.w3.org/TR/xptr/`.

Consortium, W. 2001c. Xquery 1.0: An XML query language. Technical report, WWW Consortium. www.w3.org/TR/xquery/.

Dyck, M. 2002. The GNU version of The Collaborative International Dictionary of English, presented in the Extensible Markup Language. `http://www.ibiblio.org/webster`.

Ferragina, P. and Mastroianni, A. XCDE, XML Compressed Document Engine. `http://butirro.di.unipi.it/ ferrax/xcde/xcdelib.html`.

GNU Project. 2000. Flex 2.5.4. `http://www.gnu.org/software/flex`.

GNU Project. 2003. Bison 1.875b. `http://www.gnu.org/software/bison/bison.html`.

Goldfarb, C. and Prescod, P. 1998. *The XML Handbook.* Prentice-Hall, Oxford.

Harman, D. 1995. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)* (1995), pp. 1–19. NIST Special Publication 500-207.

Jones, K. 2000. XMLGrep. `http://sources.redhat.com/ml/xsl-list/2000-07/-msg01002.html`.

Kay, M. 2002. SAXON, the XSLT Processor. `http://saxon.sourceforge.net`.

Lapp, J., Robie, J., and Schac, D. 1998. XML query language (XQL). In *QL'98 - The Query Languages Workshop* (December 1998). `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

Meier, W. 2002. eXist, Open Source Native XML Database. `http://exist.sourceforge.net`.

Microsoft Corp. 2002. MSXML, Microsoft XML. `http://msdn.microsoft.com/nhp/-?contentid=28000438`.

Navarro, G. 1995. A language for queries on structure and contents of textual databases. Master's thesis, Dept. of Computer Science, Univ. of Chile. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/thesis95.ps.gz`.

Navarro, G. and Baeza-Yates, R. 1995. A language for queries on structure and contents of textual databases. In *Proc. ACM SIGIR'95* (1995), pp. 93–101.

Navarro, G. and Baeza-Yates, R. 1997. Proximal Nodes: a model to query document databases by content and structure. *ACM TOIS 15*, 4 (Oct), 401–435.

Toronto XML Server Project. 2002. ToXin, Toronto XML Server. `http://www.cs.toronto.edu/tox`.