XQL and Proximal Nodes

Ricardo Baeza-Yates   Gonzalo Navarro

Depto. de Ciencias de la Computación, Universidad de Chile
Blanco Encalada 2120, Santiago 6511224, Chile
E-mail: {rbaeza,gnavarro}@dcc.uchile.cl *

Abstract

Despite that several models to structure text documents and to query on this structure have been proposed in the past, a standard has emerged only relatively recently with the introduction of XML and its proposed query language XQL, on which we focus in this paper. Although there exist some implementations of XQL, efficiency of the query engine is still a problem. We show in this paper that an already existing model, *Proximal Nodes*, which was defined with the goal of efficiency in mind, can be used as an efficient query engine behind an XQL front-end.

1 Introduction

Searching on structured text is becoming more important with the increasing use of XML [GP98]. Although SGML [Int86] existed for a long time, its complexity was the main limitation for a wider use. By taking advantage of the structure, content queries can be made more precise. This issue is becoming more and more important, because as the availability of textual data increases, structure and metadata can help in coping with volume explosion. Also, XML data can be seen as the meeting point between the database community (in particular the work on semi-structured data and query languages for XML) and the information retrieval community (structured text models).

Several models have been proposed since the eighties to structure text documents and to query on this structure as well as on content (see for example a survey from 1996 [BYN96]). The importance of the area has grown rapidly with the adoption of XML as a standard to structure text. After the widespread acceptance of XML, most of the research on structured text has not concentrated anymore on proposing new structuring models but on designing suitable query languages that work on XML-structured text. There is not yet a standard for querying XML. Rather, there were a number of proposals when this work was done (1999). Among those we concentrate on XQL [LRS98], which was one of the strongest candidates to become the standard, and has influenced the current unified proposal, Xquery [Con01]. At the end of 1999, also appeared Xpath [Con99], the query language for XML paths, which is inspired in XQL, and hence many of our results are also valid for Xpath.

*This work was supported by Fondecyt Project 1-990627.

Many of the models proposed to structure and query text documents had the goal of efficiency in mind. As a result, several studies on the tradeoff between query language expressiveness and the efficiency of its implementation were carried out [CM95, BYN96]. On the other hand, to the best of our knowledge there are no efficient implementations of XQL, in the sense that the structure is not fully indexed.

In 1995 we designed one of those structured text models, called *Proximal Nodes* (PN), with this tradeoff in mind [NBY95b, NBY97]. PN occupied a place in the expressiveness/efficiency graph, where there existed other more expressive and less efficient models as well as more efficient and less expressive models.

Our goal in this paper is to show that PN fits very well in the XML/XQL scheme. Its structuring model matches quite well with XML (indeed, it is a little more powerful), and its query model is just expressive enough to represent the operations required by XQL. Other structured text models proposed are either less efficient than what is possible or less expressive than what is necessary to support XQL.

This makes the PN model an excellent alternative to implement XQL. We envision a system which gives access to a XML document database and uses XQL (or another XML query language) as its front-end query language. Queries in XQL are translated into the syntax of the PN model. On the other hand, the XML database is indexed as required by the PN model implementation. Hence, the back end of the engine executes the query inside the PN model and delivers the result back to the front-end, which in turn represents the result in XML format to the end user. See Figure 1.

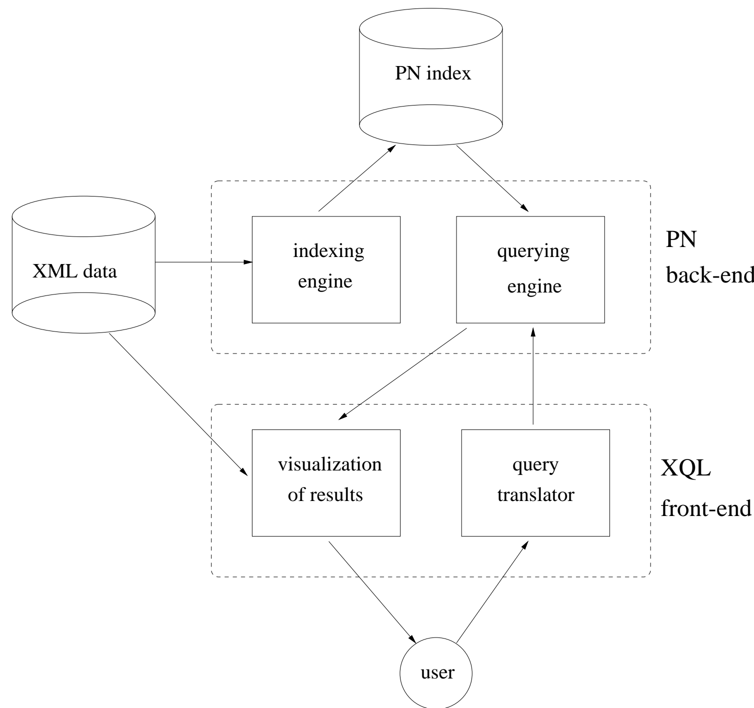

Figure 1: A XML/XQL system backed by the PN model.

This paper is organized as follows. Section 2 briefly introduces the reader to XML and its query languages, as well as to the existing structured text models. Section 3 presents the XQL query language. Section 4 presents the PN model in some depth. Section 5 shows how XQL can be implemented using the PN model. We conclude with some work in progress.

2 XML, XQL and Other Structured Text Models

XML stands for eXtensible Markup Language [GP98] and is a simplified subset of SGML [Int86], a metalanguage for tagging structured text. That is, XML is not a markup language, as HTML is, but a meta-language that is capable of containing markup languages in the same way as SGML. XML allows having human-readable semantic markup, which is also machine-readable. As a result, XML makes it easier to develop and deploy new specific markup, enabling automatic authoring, parsing, and processing of networked data.

XML does not have many of the restrictions imposed by HTML and, on the other hand, imposes a more rigid syntax on the markup, which becomes important at processing time. In XML, ending tags cannot be omitted. Also, tags for elements that do not have any content, like `"BR"` and `"IMG"`, are specially marked by a slash before the closing angle bracket. XML also distinguishes upper and lower case, so `"img"` and `"IMG"` are different tags (unlike in HTML). In addition, all attribute values must appear between quotes. That implies that parsing XML without knowledge of the tags is easier. In particular, using a DTD (data definition table, as used in SGML) is optional. If there is no DTD, the tags are obtained while the parsing is done. With respect to SGML, there are a few syntactic differences, and many more restrictions.

In 1998 the WWW Consortium requested proposals for a standard query language for XML. Proposed query languages for XML included XQL [LRS98], XML-QL [FDL+99], XGL [CCD+99], Lorel [Wid99], Ozone [LAW99], Quilt [CRF00], and Squeal [SS00]. Based on many of these languages, the WWW Consortium has recently presented Xquery [Con01], the proposed standard for XML query language. Another recently important issue is the integration of these languages with information retrieval approaches (for example see [Wid99, FMK00, SM]). There are query languages for XML which are more powerful than XQL, as well as for the Web as a database. A complete comparison of these languages is presented in this issue [LLD+].

Few implementations for XQL have been offered, partly because of the rise of newer XML query language proposals. Among these we can mention the inclusion of XQL in Software AG's Tamino, the GMD-IPSI's engine, and a subset included in Microsoft Explorer 5.0 browser[Rob01]. There is almost no documentation about the internals of these implementations, but the available ones imply that the XML structure is not fully indexed and sequential search is used in most cases.

Before XML appeared, several models to query structured text were proposed. They were designed with efficiency as one of their goals. These approaches are characterized by generally imposing a hierarchical structure on the database, and by mixing queries on content and structure. Although this structuring is simpler than, for example, hypertext, even in this simpler case the problem of mixing content and structure is not satisfactorily solved. A survey on this topic can be found in [BYN96].

Some of the most prominent examples of these models follow, together with the reasons that make them unsuitable as a back-end for XQL. The Hybrid Model proposed in [BY96] is one of

the oldest and does not permit nesting in the structural components, so hierarchies cannot be expressed. Moreover, the structure is fixed, which is too poor for XML. PAT Expressions [ST92] permit the dynamic definition of different structural components in the text, but these, as well as intermediate results, must be disjoint text areas, so one cannot make the union of chapters and sections. Overlapped lists [CCB95] permit overlapping but still not nesting. Lists of References [Mac91] permits nesting but only the top level elements are returned as the result of a query (so the union of chapters and sections returns just the chapters). Sgrep and other similar models [DSDT96, JK96] permit nesting and overlapping in the results, but they restrict the query operations to the simplest containment and proximity ones, which in insufficient for XQL. In particular they do not permit querying on positional inclusion and direct ancestorship (see later). Parsed Strings [GT87] has different goals, being a data manipulation language rather than a query language. Tree Matching [KM93] offers powerful structural matching operators but it is weak at relating text content and structure. Due to its power on structural queries, which is unnecessarily powerful for XQL, the implementation of this model is very inefficient. Newer developments on this model [MS01] obtained polynomial time solutions but they do not seem practical enough for large document databases. Recent work has focused on filtering techniques [MS99] and other extensions.

3 The XQL Query Language

We briefly describe in this section the capabilities of XQL. We do not intend to give a complete overview of XQL (see the original references [LRS98] for this sake). Rather, we aim at showing the type of queries that XQL permits through a running example which is given in Figure 2. The example is an extract of a book with a structure of chapters, and sections, where sections can contain other sections and all have titles. We also permit constructions such as figures and lists. Different constructions have different attributes. We assume that the reader is familiar enough with XML so as to understand the example without further explanation.

3.1 Path Expressions

The main type of expression in XQL is the so called "path expression", for example `"chapter/section/title"`, which retrieves all the titles that descend directly from sections that descend directly from chapters (i.e. titles of top level sections). In our example this query would return

```
<title>Motivation</title>
<title>Basic Concepts</title>
<title>How to Use this Book</title>
```

Transitive inclusion is expressed using a double bar "//", for example `"chapter//section/title"` returns titles of sections that descend from a chapter. In our example this returns

```
<title>Motivation</title>
  <title>Information versus Data Retrieval</title>
```

```
<book publisher="Addison-Wesley" isbn="0-201-39829-X">
 <author>R. Baeza-Yates</author>
 <author>B. Ribeiro-Beto</author>
 <title>Modern Information Retrieval</title>
 <chapter number="1">
  <title>Introduction</title>
  <section number="1.1">
   <title>Motivation</title>
   Information retrieval (IR) deals with the representation, ...
   <section number="1.1.1">
    <title>Information versus Data Retrieval</title>
    Data retrieval, in the context of an IR system, consists mainly of ...
   </section>
   <section number="1.1.2">
    <title>Information Retrieval at the Center of the Stage</title>
    In the past 20 years, the area of information retrieval has grown ...
   </section>
   ...
  <section number="1.2">
   <title>Basic Concepts</title>
   The effective retrieval of relevant information is directly affected ...
   <figure number="1.1" file="ir-system.eps"
    caption="Interaction of the user with the retrieval system ..."/>
  </section>
  ...
  <section number="1.6">
   <title>How to Use this Book</title>
   Although several people have contributed chapters for this book ...
   <section number="1.6.1">
    <title>Teaching Suggestions</title>
    This textbook can be used in many different areas including ...
    <list>
     <item title="Information Retrieval"> this is the standard ...  </item>
     <item title="Advanced Information Retrieval"> similar to the ...  </item>
     ...
     <item title="Digital Libraries"> this course could start ...  </item>
    </list>
   </section>
  </section>
 </chapter>
</book>
```

Figure 2: Example of a XML formatted document.

```
  <title>Information Retrieval at the Center of the Stage</title>
<title>Basic Concepts</title>
<title>How to Use this Book</title>
  <title>Teaching Suggestions</title>
```

This mechanism is extended to permit "absolute paths", i.e. paths that are evaluated from the root of the structure tree, e.g. "/book/title" corresponds to the unique element

```
<title>Modern Information Retrieval</title>
```

XQL permits also the use of the wild card where a structural name is expected, for example "book/*/title" to mean titles directly descending from something that descends directly from a book constructor. The answer is the only chapter title in the example

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

Finally, XQL permits queries of the type "section/section[2]", meaning the second (sub)section contained in a section, as well as ranges such as "chapter/section[2-3]". The first example would return

```
 <section number="1.1.2">
  <title>Information Retrieval at the Center of the Stage</title>
  In the past 20 years, the area of information retrieval has grown .. .
 </section>
<section number="1.2">
 <title>Basic Concepts</title>
 The effective retrieval of relevant information is directly affected . ..
 <figure number="1.1" file="ir-system.eps"
  caption="Interaction of the user with the retrieval system ..."/>
</section>
```

3.2 Filters

It is also possible to express that one wants the top level instead of the bottom level nodes. This is done by using filters "[]", e.g. "chapter[section/figure]", which selects chapters that contain a figure contained in a section. This returns the whole chapter in our example, since it contains a section that directly contains a figure.

To obtain chapters that directly contain a section that directly contains a figure, one would write "chapter[/section/figure]", which in our case would retrieve the same result as before since the section containing the figure is 1.2, a top level one.

Similarly, one can express "book[author[0] = "R. Baeza-Yates"]", where the condition is that the first "author" element that descends from the book equals the string "R. Baeza-Yates". The query would return the whole book.

One can combine path expressions with filters, for example "book[author = "B. Ribeiro-Neto"] /figure" would return all the figures of books written by B. Ribeiro-Neto, in our example

```
<figure number="1.1" file="ir-system.eps"
 caption="Interaction of the user with the retrieval system ..."/>
```

XQL permits boolean operations inside the filters. For example, "chapter[figure or title = "Introduction"]" returns chapters that either contain a figure or have inside a title that reads "Introduction". Similarly we could ask for chapters that meet both conditions using "chapter[figure and title = "Introduction"]". Finally, "chapter[not figure]" retrieves chapters without figures.

An XQL extension permits to say "any" or "all" inside the condition. The first does not alter the usual semantics, e.g. both "chapter[figure]" and "chapter[any figure]" retrieve chapters that contain (any) figures. The second construction, "all", makes sense for predicates of equality or inequality and requires it to hold for all the relevant constructors inside the element. For example "book[all author != "A. Moffat"]" requires that no author field inside the book be equal to "A. Moffat".

3.3 Other Features

Attributes Another widely used feature of XQL are the attributes of the nodes. Each structural node in XML can have a number of attributes, which have a name and a value; and it is possible to restrict the matches to those having some attribute and even to those where some attribute has some property. Contrary to a subfield, an attribute cannot have any internal structure.

For example "book[@publisher = "Addison-Wesley"]" selects the books whose attribute "publisher" is "Addison-Wesley", while "book[@isbn]" selects books where there is an "isbn" field. It is possible to use all the boolean operators and other filters on attribute values.

Semijoins XQL permits comparing the content of absolute paths. For example "book[@author = @me]", where "@me" is an attribute that descends directly from the root of the hierarchy.

This can be done only for absolute paths, so that the reference can have just one value across the whole collection.

Methods XQL is designed to be embedded in Perl, and as such it imports many of the Perl's functions. Some of the functions of interest are: text(), which corresponds to the textual content of a node; value(), which is similar to text() except that it can be cast to other types such as integer or float; and other aggregate functions such as the number of structural components below a given node.

Set Operations XQL permits set operations such as union, intersection and difference. For example "figure union list" returns all the figures and lists in the books.

Proximity Operations Despite that the standard XQL does not permit it, some implementations allow a kind of "followed by" operation, e.g. return sections followed by a figure. Unfortunately their description is not very clear.

4 The Proximal Nodes Model

The Proximal Nodes Model (PN) [Nav95, NBY95b, NBY97] 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, but allowing overlaps between areas delimited by different hierarchies (e.g. chapters/sections/paragraphs and pages/lines). A query can relate different hierarchies, but returns a subset of the nodes of one of them 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. In [NBY95a, Nav95], the expressiveness of this model is compared against others and found competitive or superior to most of them. 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). The time to solve a query is proportional to the sum of the sizes of the intermediate results (and not the size of the database).

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 en 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 our experiments the lazy version worked better for more complicated queries and worse for simpler queries.

The Proximal Nodes 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.

Figure 3 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

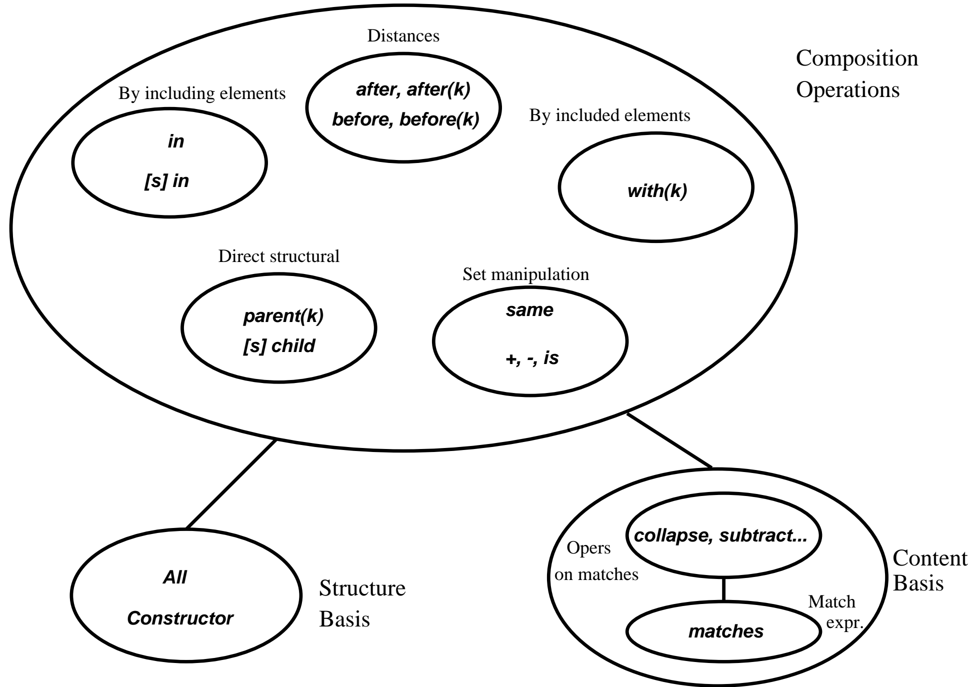

Figure 3: Possible operations for our model, classified by type. We have removed those that are relevant when several hierarchies exist, which is not the case in XML.

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, `"information"` generates the flat set of all segments of 11 letters where that word appears in the text. 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 chapters in a book.
- Name of hierarchy: ("**All**" queries). Is the set of all nodes of the hierarchy. The same effect can be obtained by summing up ("+" 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, `figure` **in** `section` selects all figures inside sections.
  - 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. We consider 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, `[3..5] section` **in** `section` retrieves the 3rd, 4th and 5th sections from all sections. If sections include other sections, only the top-level ones are 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, `section` **with**(5) `"information"` selects the sections in which the word "information" appears five or more times.

- 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.

- 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, `list` **after** `figure` (`chapter`) retrieves the nearest lists following figures, inside the same chapter.

  - "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, `"information"` **before** (10) `"retrieval"` (`section`) selects the words "information" that are followed by "retrieval" at a distance of at most 10 symbols, inside the same section.

- Set manipulation operators: Manipulate both operands as sets, implementing union, difference, and intersection under different criteria.
  - "$P + Q$" is the union of P and Q. For example, `figure` + `list` is the set of all figures and lists. To make a union on text segments, use **collapse**.
  - "$P - Q$" is the set difference of P and Q. For example, `chapter` − (`chapter` **with** `figure`) are the chapters with no figures. To subtract text segments, we resort to operations on matches.
  - "P **is** Q" is the intersection of P and Q. For example, ([1] `section` **in** `chapter`) **is** (`[3]` `section` **in** `book`) selects the sections which are first (top-level) sections of a chapter and at the same time third (top-level) section of the book. 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".

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.

The Proximal Nodes 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 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.

At query time, each node of the query syntax tree is converted into an intermediate result from the leaves to the root (other evaluation orders are considered but here we explain this one for clarity). The intermediate results are trees which are subsets of the whole hierarchy. Leaves which

are 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 a particular case of a tree of answers.

Finally, internal query nodes correspond to operations that are carried out once their operands have been solved and converted into trees of nodes. 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.

5 Implementing XQL Operations using Proximal Nodes

We show in this section how the XQL query language can be implemented using the PN model. We start by considering the XML structure and then pay attention to each XQL feature in turn. In passing, we show more in detail why other models are not suitable for implementing XQL.

5.1 Interpreting the XML Structure

The first problem when mapping XQL over the PN query language is whether the two structuring models are the same. First, PN permits several hierarchies and XML just one, so we will not use the extended PN features. Second, XML permits arbitrary hyperlinks to other parts of the structure, i.e. defining reference points and later referencing them, so that the referenced node is assumed to be duplicated at the referencing point. This permits XML building an arbitrary graph, when loops and hence infinite paths may exist. The PN model does not permit expressing this kind of structure, so we assume that references are either disregarded or physically duplicated when they do not form cycles.

Both solutions imply that PN is insufficient to express XML references. On the other hand, the only query models able to deal with this kind of structure are graph query languages such as Hy^+ [CM93], which are not efficient enough to handle large text databases. We strongly believe that no efficient XML query language can incorporate this feature.

The final feature where XML and the PN model differ is in the attributes. PN does not permit attributes to be attached to structural nodes. A lot of attention is given to attributes in the XML/XQL literature.

Let us consider again Figure 2. We have for example that the book has some attributes such as `"publisher"` and `"isbn"`, as well as some unique fields such as `"title"`. Should the publisher be an attribute or a unique field? Let us consider the sections, which have an attribute `"number"` and a unique field `"title"`. Could the number be a field? The answer is that choosing that a given piece of information is an attribute or a field is a matter of design decision, except that attributes must (1) be unique and (2) have no internal structure. Therefore, any attribute can be converted into a field. An alternative for the first lines of the example of Figure 2 is

```
<book>
 <publisher>Addison-Wesley</publisher>
 <isbn>0-201-39829-X</isbn>
 <author>R. Baeza-Yates</author>
```

```
<author>B. Ribeiro-Beto</author>
<title>Modern Information Retrieval</title>
<chapter number="1">
...
```

Hence, the indexer that reads the XML data and builds the index needed for PN will treat attributes just as fields, as any other descendant of the node. In this sense the PN model is indeed more general than XQL since it does not need to make such distinctions. The tags are used for discovering the structure of the text, but these words are not indexed as part of the content of the documents.

Figure 4 shows the hierarchical representation of the PN index for our example of Figure 2. As it can be seen, all the text regions are included in the hierarchy and the tags are used to delimit the extent of the structural nodes. Observe also how attributes and fields are treated the same way.

5.2 Path Expressions

At a first sight, the XQL query language looks rather different from the presented query language. Typical XQL expressions are of the form

`chapter/section/title`

where the "/" represents direct inclusion. However, the above expression is translated directly into a Proximal Nodes query

`title` **child** `section` **child** `chapter`

that is, the lowest level elements are selected. Despite that it looks as a navigational operation (i.e. enter into chapters, then move to sections, then to titles), we can regard it as a search operation for nodes of a certain type and certain ancestors.

The "/" operation is the most basic one in XQL, and it immediately outrules many alternative models to query structured text based on positional information only, since they cannot query by direct ancestorship. Transitive inclusion can also be expressed using a double bar "//", and it can be translated into the **in** operation in Proximal Nodes.

The most navigational-looking feature of XQL is its ability to express absolute paths, i.e. paths that are evaluated from the root of the structure tree. This can be simulated by adding an extra single root R node to the hierarchy and adding "**child** R" to those queries.

The use of wild cards for structural names is permitted in XQL, so one can write `"book/*/title"` to mean titles directly descending from something that descends directly from a book. The wild card can be replaced by the Proximal Nodes feature that permits using "**All**" as a node name, whose result is the whole hierarchy.

XQL permits queries of the type $X/Y[k]$, meaning the k-th Y contained in each X. This corresponds exactly to the positional inclusion feature of Proximal Nodes, which cannot be found in any other existing model. In both models this can be extended to arbitrary ranges of values, and to indices relative to the first or to the last included element.

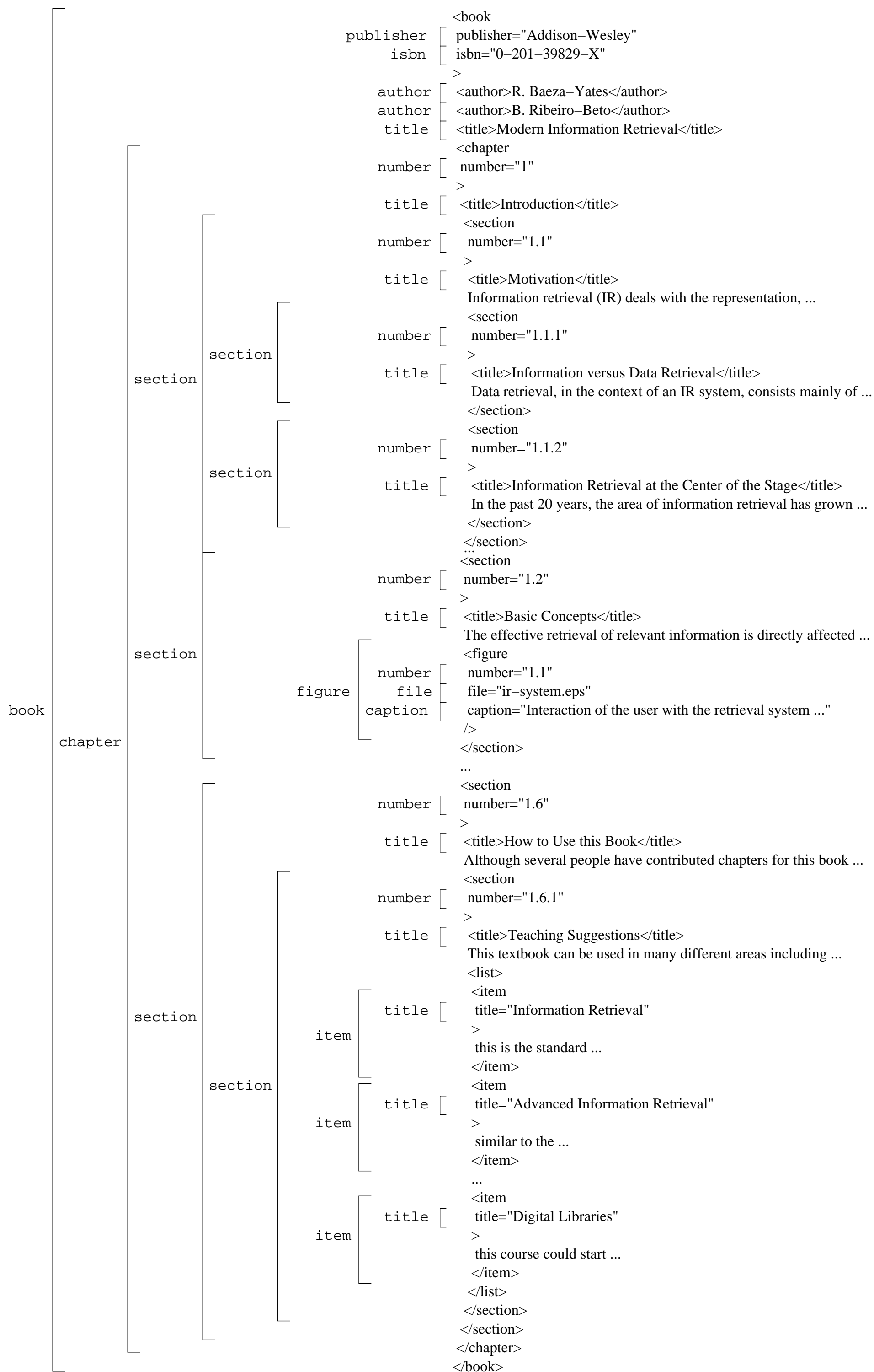

Figure 4: Structural representation of our example, as it is stored by the PN index.

5.3 Filters

It is also possible to express that one wants the top level instead of the bottom level nodes. This is done by using filters "[]", e.g.

$$\texttt{chapter}[\texttt{section/figure}]$$

which selects chapters that contain a figure contained in a section. This is easily translated into

$$\texttt{chapter}\ \textbf{with}\ (\texttt{figure}\ \textbf{child}\ \texttt{section})$$

Similarly, one can express "book[author[0] = "R. Baeza-Yates"]", where the condition is on the first "author" element that descends from the book. This can be translated into positional inclusion as

$$([1]\ \texttt{author}\ \textbf{in}\ \texttt{book})\ \textbf{same}\ \text{"}\texttt{R. Baeza} - \texttt{Yates}\text{"}$$

XQL permits boolean operations inside the filters, and this requires more care. First, "$X[Y\ or\ Z]$" (which selects the X elements that contain some Y or some Z element) can be converted into $X[Y\ +\ Z]$. On the other hand, "$X[Y\ and\ Z]$" requires that X contains both some Y and some Z, which can be converted into $(X[Y])[Z]$. Finally $X[not\ Y]$, which selects the X elements containing no Y element, can be rewritten as $X - X[Y]$, although Proximal Nodes permits the negated variants of the containment operations (e.g. **not with**), and this has a more efficient implementation.

An XQL extension permits to say *any* or *all* inside the condition. While *any* maintains the normal semantics, *all* requires extra care. For example, "book[all author != "A. Moffat"]" requires that no author field inside the book be equal to "A. Moffat". This cannot be directly expressed in Proximal Nodes but it can be converted using double negation: $X[all\ Y] = X - X[not\ Y]$.

5.4 Attributes

Another widely used feature of XQL are the attributes of the nodes. This seems to deviate significantly from simplifying models. Each structural node can have a number of attributes, which have a name and a value; and it is possible to restrict the matches to those having some attribute and even to those where some attribute has some property. For example

$$\texttt{book}[@\texttt{publisher} = \text{"}\texttt{Addison} - \texttt{Wesley}\text{"}]$$

selects the books whose attribute "publisher" is "Addison-Wesley". The key observation is that an attribute appears in the text inside the text region of its node and is clearly identified by its name. Hence, as explained, it is not hard for the indexer to identify it and to treat it just as any other descendant of the node. The previous query can be thus translated into

$$\texttt{book}\ \textbf{parent}\ (\texttt{publisher}\ \textbf{same}\ \text{"}\texttt{Addison} - \texttt{Wesley}\text{"})$$

where "Addison-Wesley" is a content query that will return all the text segments where that string appears, and "publisher" will return all the text areas that correspond to "publisher" attributes.

Their intersection yields precisely the desired result. Note that we have treated the attribute as a normal field. In this sense the Proximal Nodes model is indeed more general than XQL since it does not need to make such distinctions. For example, XQL treats as a different operation the query for structural elements whose text value is equal to some constant, while in Proximal Nodes this is exactly the same query we have just considered.

5.5 Semijoins

A somewhat special feature allowed by XQL is to permit taking as constants the content of absolute paths. For example `"book[@author = @me]"`, where `"@me"` is an attribute that descends directly from the root. This is not contemplated in the Proximal Nodes model, but can be easily fixed at the query processing phase, by detecting such cases, getting the text directly from the file, and replacing the reference by the text constant.

The key issue is that this can be done only for absolute paths, so that the reference can have just one value. The generalized feature is called a "semijoin" and is not supported neither by XQL nor by Proximal Nodes. The semijoin would allow selecting chapters whose title is mentioned in the bibliography section of a given book. Note that this violates the condition of Proximal Nodes model: the fact that a book qualifies or not cannot be determined considering the text areas of the titles of the books, but one needs to compare the content of the titles with those found in the bibliography of the other book. This is hard to implement efficiently [CM95, BYN96].

5.6 Methods

As XQL is embedded in Perl, it imports many of Perl's functions. Of course this is not general and cannot be expected to be supported by an abstract model such as Proximal Nodes. However, the most used methods are indeed supported. First, `text()` corresponds to the textual content of a node, which is a basic method for Proximal Nodes. Second, `value()` is similar to `text()` but it can be cast to other types such as integer or float. This permits putting, say, numerical conditions on the content of a given attribute. Despite that this cannot be solved by a text index, the Proximal Nodes model can coexist with many independent content indexes, and therefore a different index able to answer such questions could be built on the numerical values found in the text and it could gracefully coexist with the rest of the system [NBY97]. This index should receive a condition, say "≤ 30", and return all the text areas containing numbers smaller than 30.

Proximal Nodes permits some conditions on XQL aggregate functions as well, such as selecting elements including (directly or transitively) at least k elements of some kind.

5.7 Set Operations

Finally, XQL permits set operations, which are directly translated into Proximal Nodes operations. It is interesting to mention that XQL requires the answer to be the set of structural nodes that satisfy the query. This also matches with the Proximal Nodes semantics, while some other models return only top-level or bottom-level elements.

5.8 Followed By

Although not included in the XQL proposal, some implementations of XQL permit a kind of "followed by" operation. Unfortunately their description is not very clear, and this is not a coincidence. As shown in [CM95], the semantics of a "followed by" operation is problematic for many models.

In Proximal Nodes this has been carefully designed so that either the element preceding the other or the element following the other are selected. However, it is not possible later to operate the result to know, say, which is the smallest X containing Y followed by Z. Once we have selected the Y's that are followed by Z, we lose information on which was the Z that made each Y to be selected, and therefore we cannot guarantee that the smallest X that contains the selected Y will also contain the corresponding Z. The Proximal Nodes model tries to fix the problem by permitting, at the moment of executing the operation, to specify X, so that we force that the smallest X that contains Y must contain Z. This, however, does not totally solve the problem.

An alternative solution is to return a "supernode" that has the necessary extension to contain Y and Z. However, this node is fake and does not fit well in the hierarchy, which yields consistency problems for later operations. The models that are able to handle this well [CCB95, JK96] do not rely on a strict hierarchy of nodes but permit their overlapping. Those models, for example, cannot cope with direct inclusion.

As shown in [CM95], it is quite difficult to find a satisfactory and consistent definition of a "followed by" operation in a hierarchical model.

A reasonable solution when there is just one hierarchy is to return the lowest level nodes that contain Y followed by Z. In this case we can later obtain nodes of some set which also contain the pair (Y, Z) by asking which of them contain some of those lowest level nodes. Of course those lowest level nodes can be larger than desired, but they still have this useful minimality property.

6 Conclusions

We have considered the problem of efficiently implementing XQL. In the best of our knowledge, current implementations could not cope with large document databases. Given that there has been considerable research in the past on efficiently implementing structured text query languages, it is natural to study which of the models already created are suitable for mapping XQL queries onto it. From the existing models we are aware of, the Proximal Nodes (PN) model seems to be the one that fits best. We have shown how to efficiently map XQL query onto PN operations. We believe that other XML query languages can also be implemented efficiently following the same ideas of this paper, including languages that are more expressive than XQL such as Xquery. Other features such as XML references, on the other hand, are probably hard to deal with efficiently.

Based on the results of this paper, we are currently designing a prototype for Xquery, using PN as the underlying model. This software is able of parsing XML in order to obtain the structural information, using that information to build a PN index, applying the PN query algorithms (simplified to the case of a single hierarchy) and displaying the results. Several issues, such as algebraic query optimization and design of efficient access plans given the query tree, are open and subject of future work.

References

[BY96] R. Baeza-Yates. An extended model for full-text databases. *Journal of Brazilian CS Society*, 3(2):57–64, April 1996.

[BYN96] R. Baeza-Yates and G. Navarro. *Integrating contents and structure in text retrieval. ACM SIGMOD Record*, 25(1):67–79, March 1996. ftp://sunsite.dcc.uchile.cl/-pub/users/gnavarro/sigmod96.ps.gz.

[CCB95] C. Clarke, G. Cormack, and F. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 1995.

[CCD+99] S. Ceri, A. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and T. Letizia. XML-GL: a graphical language for querying and restructuring XML documents. In *WWW8*, 1999.

[CM93] M. Consens and A. Mendelzon. Hy+: A hygraph-based query and visualization system. In *Proc. ACM SIGMOD'93*, pages 511–516, 1993. Video presentation summary.

[CM95] M. Consens and T. Milo. Algebras for querying text regions. In *Proc. PODS'95*, 1995.

[Con99] WWW Consortium. Xpath 1.0: XML path language. Technical report, WWW Consortium, 1999. www.w3.org/TR/xpath.html.

[Con01] WWW Consortium. Xquery 1.0: An XML query language. Technical report, WWW Consortium, 2001. www.w3.org/TR/xquery/.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.

[DSDT96] T. Dao, R. Sacks-Davis, and J. Thom. Indexing structured text for queries on containment relationships. In *Proc. of the 7th Australasian Database Conference*, 1996.

[FDL+99] D. Florescu, A. Deutsch, A. Levy, D. Suciu, and M. Fernandez. A query language for XML. In *Eighth International World Wide Web Conference*, 1999. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.

[FMK00] D. Florescu, I. Manolescu, and D. Kossmann. Integrating keyword search into XML query processing. In *WWW9*, Amsterdam, May 2000.

[GP98] C. Goldfarb and P. Prescod. *The XML Handbook*. Prentice-Hall, Oxford, 1998.

[GT87] G. Gonnet and F. Tompa. Mind Your Grammar: a new approach to modelling text. In *Proc. VLDB'87*, pages 339–346, 1987.

[Int86] International Standards Organization. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, 1986. ISO 8879-1986.

[JK96] J. Jaakkola and P. Kilpeläinen. Using sgrep for querying structured text files. Technical Report C-1996-83, Dept. of Computer Science, Univ. of Helsinki, Finland, 1996. Software available at `http://www.cs.helsinki.fi/u/jjaakkol/sgrep.html`.

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proc. ACM SIGIR'93*, pages 214–222, 1993.

[LAW99] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Seventh International Workshop on Database Programming Languages*, Kinloch Rannoch, Scotland, September 1999.

[LLD+] R. Luk, H.V. Leong, T. Dillon, A. Chan, W.B. Croft, and J. Allan. Improving index structures for structured document retrieval. *JASIS*. This issue.

[LRS98] J. Lapp, J. Robie, and D. Schac. XML query language (XQL). In *QL'98 - The Query Languages Workshop*, December 1998. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[Mac91] I. MacLeod. A query language for retrieving information from hierarchic text structures. *The Computer Journal*, 34(3):254–264, 1991.

[MS99] H. Meuss and C. Strohmaier. Improving index structures for structured document retrieval. In *21st Annual Colloquium on IR Research (IRSG'99)*, 1999.

[MS01] H. Meuss and K. Schulz. Complete answer aggregates for tree-like databases: A novel approach to combine querying and navigation. *ACM Transactions on Information Systems*, 2001. To appear.

[Nav95] G. Navarro. A language for queries on structure and contents of textual databases. Master's thesis. Dept. of Computer Science, Univ. of Chile. `ftp://-sunsite.dcc.uchile.cl/pub/users/gnavarro/thesis95.ps.gz`, 1995.

[NBY95a] G. Navarro and R. Baeza-Yates. Expressive power of a new model for structured text databases. In *Proc. PANEL'95*, pages 1151–1162, 1995. `ftp://-sunsite.dcc.uchile.cl/pub/users/gnavarro/clei95.ps.gz`.

[NBY95b] G. Navarro and R. Baeza-Yates. A language for queries on structure and contents of textual databases. In *Proc. ACM SIGIR'95*, pages 93–101, 1995. `ftp://-sunsite.dcc.uchile.cl/pub/users/gnavarro/sigir95.ps.gz`.

[NBY97] G. Navarro and R. Baeza-Yates. Proximal Nodes: a model to query document databases by content and structure. *ACM TOIS*, 15(4):401–435, Oct 1997.

[Rob01] J. Robie. XQL FAQ, 2001. www.ibiblio.org/xql/.

[SM] Torsten Schlieder and Holger Meuss. Querying and ranking xml documents. *JASIS*. This issue.

[SS00] E. Spertus and L.A. Stein. Squeal: A structured query language for the Web. In *WWW9*, Amsterdam, May 2000.

[ST92] A. Salminen and F. Tompa. PAT expressions: an algebra for text search. In *COMPLEX'92*, pages 309–332, 1992.

[Wid99] J. Widom. Data management for XML: Research directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, 1999.