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

We consider the recently proposed XQL language, which is designed to query XML documents by content and structure. We show that an already existing model, namely "Proximal Nodes", is the only one that addresses all the complex querying operations defined by XQL and that suggests an efficient implementation for them.

1 Introduction

Searching on structured text is becoming more important with the increased use of XML. Although SGML 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. 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) with the information retrieval community (structured text models). Our main goal in this paper is to show the similarity of both approaches.

In 1995 we published a survey on structured text models [BYN96] where we envisioned the importance of this area. Today, XML makes these models even more important, because as the availability of textual data increases, structure and metadata can help in coping with volume explosion. Most structured text models are based on hierarchies, typically based on trees. XML naturally adapts to that type of models. More over, existing structured text models can be used to implement efficiently the proposed query languages for handling XML based data. In particular, we show how the Proximal Nodes (PN) model [NBY95b, NBY97] can be used to efficiently implement the XQL query language[LRS98].

Section 2 briefly introduces the reader to XML and its query languages, as well as known structured text models. Section 4 presents the PN model and the operations that supports. Section 5 shows how XQL matches the PN model and we conclude with some work in progress.

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

2 Basic Concepts

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 to have 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 (this is not true in HTML). In addition, all attribute values must be between quotes. That implies that parsing XML without knowledge on the tags is easier. In particular, using a DTD 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.

Recently, the WWW Consortium requested proposals for a standard query language for XML. XML query languages include XQL [LRS98], XML-QL [DFF$^+$98], XGL [CCD$^+$99], Lorel [Wid99], Ozone [LAW98], and Squeal [SS00]. Also, there are query languages for XML which are more powerful than XQL, as well as, for the Web as a database. Recently, another important issue is the integration of these languages with information retrieval approaches (for example see [Wid99, FMK00]).

Before XML appeared, several models to query structured text were proposed. 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.

The models include (in increasing expressiveness order) the Hybrid Model [BY96], PAT Expressions [ST92], Overlapped Lists [CCB95], Lists of References [Mac91], Proximal Nodes [Nav95, NBY95b, NBY97], Parsed Strings [GT87], and Tree Matching [KM93]. In most cases, the time complexity of queries has a trade-off with the expressiveness of the model.

The characteristics of these models are summarized in Table 1, and in this work we focus on the PN model, which has a good balance between expressiveness and time efficiency.

3 Operations Supported by the Proximal Nodes Model

The Proximal Nodes Model [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

Model	Structuring mechanism	Contents query language	Structural query language
Hybrid Model [BY94]	IR-like documents + fields + text. Fields can nest and overlap, but this cannot be later queried. It is a flat model.	Query = matches + documents. Almost all the language is oriented to matches, which are seen as their start point. Expresses distances. Has separate set manipulation tools for matches and documents.	Only to restrict match points to be in a given field or to select fields including match points (selected fields are then seen as match points). Very simple in general.
PAT Expressions [ST92]	Dynamic definition of regions, by pattern matching. Each region is a flat list of disjoint segments.	Powerful matching language. Handles points and regions. Has set manipulation operations. Expresses distances.	Simple, since structures are flat. Can express inclusion, set manipulation and some very specialized operations.
Overlapped Lists [CCB95]	A set of regions, each one a flat list of possibly overlapping segments.	Not specified. Words and regions are seen in a uniform way, by an inverted list metaphor.	Results can overlap, but not nest. Can express inclusion, union and combinations.
Lists of References [Mac91]	A single hierarchy with attributes in nodes and hypertext links.	Text queries can only be used to restrict other queries.	Results are flat and from the same constructor. Can express inclusions, complex context conditions and set manipulation.
Proximal Nodes [NBY95b]	A set of disjoint strict trees (views), Views can overlap. Nodes cannot be dissociated from segments.	Text is a special view. Text queries are leaves of query syntax trees. Text content is accessed only in matching subqueries, thereafter it is seen just as segments. There are powerful distance operators, and special set operators for text.	Can express inclusion, positions, direct and transitive relations, and set manipulation. Can express complex context conditions if they involve proximal nodes.
Tree Matching [KM93]	A single tree, with strict hierarchy. No more restrictions.	Not specified, orthogonal to the model. It can only be used to restrict sets of nodes of the tree (leaves of patterns). Weak link between content and structure.	Powerful tree pattern matching language. Can distinguish order but not positions nor direct relationships. Can express equality between different parts of a structure, by using logical variables. Set manipulation features via logical connectives.

Table 1: Features of structured models.

them only (i.e. nested elements are allowed in the answers, but not overlaps). Each node has an associated segment, 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, being linear 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). A lazy version is also studied, which behaves better in some situations. This model is as efficient as many others which are less expressive.

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 1 shows the schema 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, `"computer"` generates the flat set of all segments of eight 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.

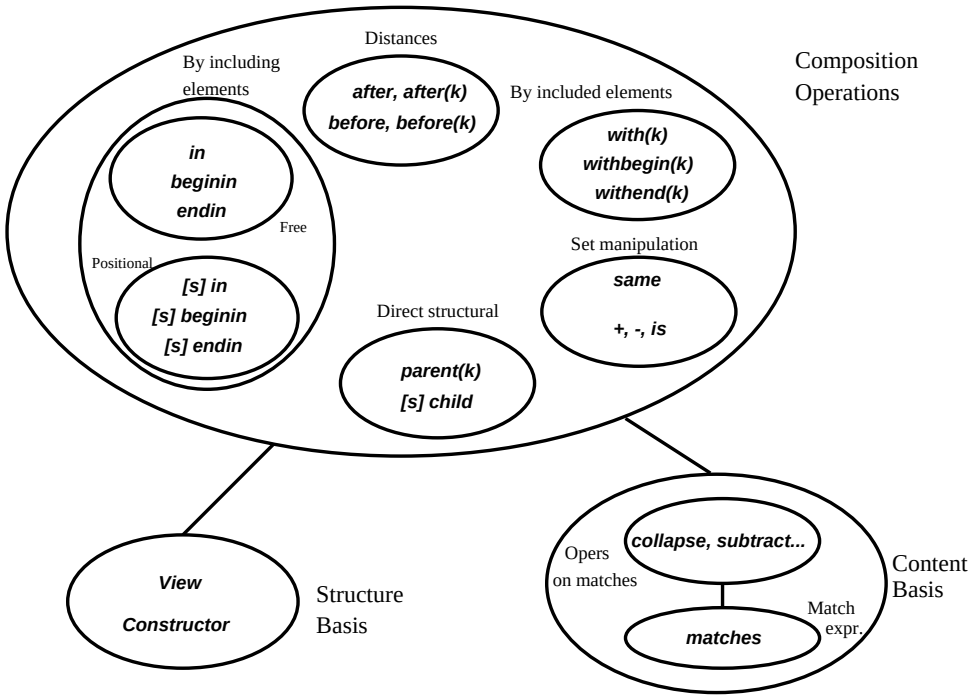

Figure 1: Possible operations for our model, classified by type.

- Basic structure operators: Are the other kind of leaves of the query syntax tree, which refer to basic structural components.
  - Name of structural component: ("**Struct**" queries). Is the set of all nodes of the given type. For example, `chapter` retrieves all chapters in a book.
  - Name of hierarchy: ("**Hierarchy**" queries). Is the set of all nodes of the given 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 in some sense 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, `citation` **in** `table` selects all citations made from inside a table.
  - 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. In order to linearize the position, for each node of Q only the top-level nodes of P not disjoint with the Q-node are

considered, and those which overlap are discarded, along with their descendants. 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]` `paragraph` **in** `page` retrieves the 3rd, 4th and 5th paragraphs from all pages. If paragraphs included other paragraphs, only the top-level ones would be considered, and those partially included in a page would be discarded (along with their subparagraphs).

- Including operators: Select from the first operand the elements including in some sense 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) `"computer"` selects the sections in which the word "computer" 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 its 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, `table` **after** `figure` (`chapter`) retrieves the nearest tables 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, `"computer"` **before** (10) `"architecture"` (`paragraph`) selects the words "computer" that are followed by "architecture" at a distance of at most 10 symbols, inside the same paragraph. Recall that this distance is measured in the filtered file (e.g. with markup removed).

- 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, `small` + `medium` + `large` is the set of all size-changing commands. 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** `page`) selects the sections which are first (top-level) sections of a chapter and at the same time third (top-level) section of a page. 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".

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 suggests 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 we explain this one for clarity). The intermediate results are trees that 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 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 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.

4 Implementing XQL Operations

4.1 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/paragraph
```

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

`paragraph` **child** `section` **child** `chapter`

that is, the lowest level elements are selected. Despite that they look as a navigational operation (i.e. enter into chapters, then move to sections, then to paragraphs), 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 the queries.

The use of wild cards for structural names is permitted in XQL, so one can write `"book/*/section"` to mean sections 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 $X/Y[3]$, meaning the third 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.

4.2 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/paragraph]`

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

`chapter` **with** (`paragraph` **child** `section`)

Similarly, one can express `"book[author[0] = John Silver]"`, where the condition is on the first "author" element that descends from the book. This can be translated into positional inclusion.

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 $X[Y \bigcup Z]$. 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**).

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!=Bob]"` requires that no author field inside the book be equal to "Bob". This cannot be directly expressed in Proximal Nodes but it can be converted using double negation: $X[$*all* $Y] = X - X[$*not* $Y]$.

4.3 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

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

selects the books whose attribute "publisher" is Addison-Wesley. A lot of attention is given to attributes. The key observation is that an attribute appears in the text inside the text region of its node and clearly identified by its name. Hence, it is not hard for the indexer to identify it and to treat it just as any other descendant of the node. The most distinguishing feature of attributes is indeed a restriction: an attribute cannot have internal structure. The previous query can be thus translated into

$$\texttt{book}\ \textbf{with}\ (\texttt{publisher} = \texttt{"Addison} - \texttt{Wesley"})$$

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.

4.4 Semijoins

A somewhat special feature permitted 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.

4.5 Methods

XQL is designed to be embedded in Perl, and as such it imports many of the 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()`,

which is similar to `text()` but it can be casted 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 context 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. 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 aggregate functions as well, such as selecting elements including (directly or transitively) at least n elements of some kind.

4.6 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.

4.7 Followed By

Despite that the standard XQL does not permit it, some implementations allow 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 loose 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] 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.

5 Conclusions

We have presented work in progress for efficiently implementing XQL. Further work on this topic will be jointly carried out with Tatiana Coello, Berthier Ribeiro-Neto, and Altigran da Silva from the Federal Univ. of Minas Gerais at Belo Horizonte, Brazil. For example, several issues about algebraic query optimization and design of efficient access plans given the query tree are open.

We believe that other XML query languages can also be implemented efficiently following the same ideas of this paper, including languages that have more expressiveness than XQL.

References

[BY94] R. Baeza-Yates. An hybrid query model for full text retrieval systems. Technical Report DCC-1994-2, Dept. of Computer Science, Univ. of Chile, 1994. `ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/hybridmodel.ps.gz`.

[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.

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

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Technical report, W3C, August 1998. `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.

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

[LAW98] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. Technical report, 1998.

[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.

[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.

[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.