Proximal Nodes: A Model to Query Document Databases by Contents and Structure

GONZALO NAVARRO and RICARDO BAEZA-YATES
University of Chile

A model to query document databases by both their content and structure is presented. The goal is to obtain a query language which is expressive in practice while being efficiently implementable, features not present at the same time in previous work. The key ideas of the model are a set-oriented query language based on operations on nearby structure elements of one or more hierarchies, together with content and structural indexing and bottom-up evaluation. The model is evaluated regarding expressiveness and efficiency, showing that it provides a good trade-off between both goals. Finally, it is shown how to include in the model other media different from text.

Categories and Subject Descriptors: H.1.2 [**Models and Principles**]: User/Machine Systems — Human information processing; H.2.1 [**Database Management**]: Logical Design — Data models; H.2.2 [**Database Management**]: Physical Design — Access methods; H.2.3 [**Database Management**]: Languages — Query languages; H.2.4 [**Database Management**]: Systems — Query processing; H.3 [**Information Storage and Retrieval**]; I.7.2 [**Text Processing**]: Document Preparation — Format and notation; Languages and systems; Standards; I.7.3 [**Text Processing**]: Index Generation

General Terms: Algorithms, Design, Human Factors, Languages, Performance

Additional Key Words and Phrases: Structured text, hierarchical documents, expressivity and efficiency of query languages, text algebras

1. INTRODUCTION

Document databases are deserving more and more attention, due to their multiple applications: digital libraries, office automation, software engineering, automated dictionaries and encyclopedias, etc. [Frakes and Baeza-Yates 1992]

The purpose of a document database is to store documents, structured or not. A document database is composed of two parts: content and (if present) structure. The content is the data itself, while the structure relates different parts of the database by some criterion.

Any information model for a document database should comprise three parts: data, structure, and query language. It must specify how is the data seen (i.e. image formats, character set, etc.), the structuring mechanism (i.e. markup, index structure, etc.), and the query language (i.e. what things can be asked, what the

Author's address: Depto. de Ciencias de la Computación, Universidad de Chile, Blanco Encalada 2120, Santiago, Chile. Email: {gnavarro,rbaeza}@dcc.uchile.cl.
This work has been supported in part by grants FONDECYT (Chile) 1940271 and 1950622.
A preliminary partial version of this work appeared in the Proceedings of SIGIR'95.

answers are, etc.).

The problem of retrieving information from document databases is normally concentrated on the text part. Text is not a relational table [Codd 1983], in which the information is already formatted and meant to be retrieved by a "key". The information is there, but there is no easy way to extract it. The user must specify what he/she wants, see the results, then reformulate the query, and so on, until satisfied with the answer. Anything that helps users to find what they want is worth considering.

Traditionally, textual databases are searched by their contents (words, phrases, etc.) or by their structure (e.g. by navigating through a table of contents), but not both at the same time. Recently, many models have appeared that allow mixing both types of queries.

Mixing contents and structure allows posing very powerful queries, being much more expressive than each mechanism by itself. By using a language that integrates both types of queries, the retrieval quality of document databases can be improved.

Suppose, for example, a software development environment with a syntax-directed editor that allows users to search all procedures that use a given global variable without assigning it, or all points where a given variable is assigned, or procedures that invoke a function defined in a given module, etc. Another example is to search in a digital library books with population statistics graphs, or where there are many illustrations regarding birds, or where an oil company is mentioned in a historical context. These queries mix content and structure of the database, and only new models can handle it.

Because of this, we see these models as an evolution from the classical ones. These new models are not fully satisfactory, though. They are not in general as mature as the classical ones. Not only they lack the long process of testing and maturing that traditional models have enjoyed, but also many of them are primitive as software systems, having been implemented mainly as research prototypes.

There are a number of challenges to be faced. On one hand, the "content" of the database is not formatted, but in natural language form. This means that no traditional model relying on formatted data (e.g. the relational model) or assuming uninterpreted data objects and relying only on their (formatted) attributes (e.g. classical multimedia databases [Bertino et al. 1988]) is powerful enough to represent the wealth of information contained in text. The information has to be extracted from the text, but not in a rigid way (Sacks-Davis, Arnold-Moore, and Zobel [1994] also argue in the same lines).

On the other hand, there is no consensus on how the structuring model of a database should be. There are a number of possible models, ranging from no structuring at all to complex interrelation networks. Deciding to use a structuring model involves choosing also what kind of queries about structure can be posed.

Finally, there is no consensus on how powerful a model should be. The more powerful the model, the less efficiently it can be implemented. We pay special attention to this expressiveness/efficiency trade-off, since being weak in either of these two aspects makes the model impractical for many applications.

The aim of this paper is to present a model to structure and query document databases, following the new trend of mixing content and structure in queries. The model is shown to be expressive and efficiently implementable. There is not at this time, to the best of our knowledge, any approach satisfying both goals (see, however, a recent work [Dao et al. 1996]). We first concentrate on text and later show how to integrate other types of data (e.g. audio, images, etc.). These media are becoming more and more common in document databases.

It has been argued that is better to put a layer integrating a traditional database system with a textual one, than trying to design a language comprising all the features [Sacks-Davis et al. 1994]. Each subsystem focuses on a different part of the query (e.g. integration of an object-oriented database with a structured text engine [Consens and Milo 1994]).

We rely on this approach. We design a language which is focused on exploiting structure- and content-related features. Other features, such as tuples and joins, should be added by integrating this language with another one oriented to that kind of operations, e.g. a relational database.

We point out what are we *not* covering in this work.

First, we do not cover languages to describe document structure, such as SGML [International Standards Organization 1986], DSSSL [International Standards Organization 1994], SPDL [International Standards Organization 1991], HyTime [International Standards Organization 1992], etc. We cover structuring models (e.g. hierarchical). A given structuring model may or may not be expressed using a given language to describe structure.

Second, we concentrate more on querying than on indexing. Although we describe different implementation alternatives for the index, we consider updates much less frequent than queries. The efficiency comparison against other models is centered on querying.

Third, we do not describe a fixed query language, but a model into which we show that a number of operations can be expressed. These include widely accepted primitives as well as new ones. The syntax we use for the query operations is not necessarily intended for final users, rather it is an operational algebra onto which one can map a more user-oriented query language.

Fourth, for several reasons, we do not make an experimental performance comparison between our model and previous work. The main problems are that for most cases the code is not available, that the performance depends heavily on the implementation (we would be comparing algorithms instead of models), that the models impose different structures on the text and retrieve different types of elements, and that such study needs a theoretical framework that does not exist (still) for structured text databases.

Finally, we do not address the important issue of merging structural queries with those involving operations such as relevance ranking (e.g. the sections or titles where the word "computer" is relevant). The reason is that even the simple problem with no relevance considerations is not yet well solved, and an integration between structuring and relevance ranking must be seen as the next goal (see [Sacks-Davis et al. 1994] and [Arnold-Moore et al. 1995] for some ideas on this problem).

This paper is organized as follows. In section 2, related work is reviewed. In section 3, our model is presented, in terms of the data model and the operations allowed for queries. In section 4 the resulting expressiveness is evaluated. In section 5 we outline the most relevant implementation aspects, and formally analyze efficiency. In section 6 we present a software architecture based on our model. In section 7 we show an experimental evaluation. In section 8 we extend our model to include other types of data. In section 9, our conclusions and future work directions are outlined. A formal syntactic and semantic definition of the model is presented in Appendix A.

Partial earlier versions of this work can be found in [Navarro and Baeza-Yates 1995b; Navarro 1995].

2. RELATED WORK

In this section we briefly review previous approaches to the problem of structuring and querying a textual database. We first mention the traditional ones, and then cover novel ideas. For a complete survey, see [Baeza-Yates and Navarro 1996].

2.1 Traditional Approaches

There are many classical approaches to the problem of querying a textual database. Some of them are: attempts to adapt the relational model [Codd 1983] to include text management [Stonebraker et al. 1983; Desai et al. 1986]; the many traditional models of information retrieval (e.g. the boolean model, the probabilistic model, the bit-vector model, the full-text model, etc.) [Salton and McGill 1983; Frakes and Baeza-Yates 1992]; hypertext [Conklin 1987] and semantic networks [Hull and King 1987; Tague et al. 1991]; and object-oriented databases [Kim and Lochovski 1989; Cattell 1991] adapted to manage text [Christophides et al. 1994].

None of these approaches satisfy our goals of mixing structure and content in queries (see, e.g. [Sacks-Davis et al. 1992]). The relational model does not adapt well to manage text, since it clearly separates a structure and a content inside the structure, and this is not the case of structured text. Classical information retrieval allows little structuring (normally only plain records and fields). Hypertexts are mostly navigational and oriented to the (network) structure. Semantic networks model the unformatted data (text, images, etc.) as a set of attributes and facts derivable from them, which is reasonable good for images but very poor for text, which brings a lot of information in its content. Finally, object-oriented databases can express the structure in a natural way, but their facilities to handle text are limited and must be implemented ad-hoc. Moreover, path expressions (which are to be used to extract structural components) are more general than the structure of documents, and therefore are less amenable of optimization (see, e.g. [Consens and Milo 1994] for an example of optimizing path expressions for the particular case of inclusion semantics). Finally, object-oriented databases are record-oriented rather that set-oriented, which is a drawback (since it forces more operational data manipulation, reminiscent of earlier navigational systems) [Date 1995].

Although those models are not powerful enough to extract the information we want from document databases, they address different problems that pure models oriented to structure do not address in general (e.g. tuples and joins, attributes, relevance ranking, etc.). We do not compare our model to those, because they address different goals.

2.2 Novel Approaches

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.

We present a sample of novel models, which cover many different approaches.

The Hybrid Model [Baeza-Yates 1996]: models a textual database as a set of documents, which may have *fields* (named areas inside records). Those fields need not to cover all the text of the document, and can nest and overlap. The query language is an algebra over pairs (D, M), where D is a set of documents and M is a set of *match points* (a text position that matches the searched word or pattern) in those documents. There is a number of operations for obtaining match points: prefix search, proximity, etc. There are operations for set manipulation of both documents and match points; for restricting matches to only some fields; and for retrieving fields owning some match point. Inclusion relationships can only be queried with respect to a field and a match point, thus the language is not fully compositional. This model can be implemented very efficiently. The original proposal for this model can be found in [Baeza-Yates 1994].

PAT Expressions [Salminen and Tompa 1992]: sees only match points, which are used to define *regions*. Regions are defined by pattern-matching expressions that specify how their endpoints look like. Each region represents a set of disjoint *segments*. A *segment* is a contiguous portion of the text. This allows dynamic definition of regions, and to translate all queries on regions to queries on matches. The need to avoid overlapping segments in regions causes a lot of trouble and lack of orthogonality in the model. This model, has been implemented very efficiently [Fawcett 1989]. The cost of this efficiency is its restrictions, which for some applications are reasonable.

Overlapped Lists [Clarke et al. 1995]: solves the problem of PAT expressions in an elegant way, by allowing overlaps, but not nesting. Each region is a list of (possibly overlapping) segments, originated by textual searches or by named regions (like chapters, for example). The idea is to unify both searches by using an extension of inverted lists, where regions and words are indexed in the same way. The implementation of this model can be as efficient as that of PAT expressions.

Lists of References [MacLeod 1991]: is a general model to structure and query textual databases, including also hypertext-like linkages, attribute management and external procedures. The structure of documents is hierarchical (no overlaps), but answers to queries cannot nest (only the top-level elements qualify), and all elements must be from the same type (e.g. only sections, or only paragraphs). Answers to queries are seen as lists of "references" (i.e. pointers to the database). This allows integration in an elegant way of answers to queries and hypertext links, since all are seen as lists of references. This model is very powerful, and because of this, hard to implement efficiently. To make the model suitable for comparison, we consider only the portion related to querying structures (even this portion is quite powerful).

Parsed Strings [Gonnet and Tompa 1987]: is in fact a structure manipulation language. A context-free grammar is used to express database schemas, that is, the database is structured by giving a grammar to parse its text. The fundamental data structure is the *p-string*, or parsed string, which is composed of a derivation tree plus the underlying text. The manipulation is carried out via a number of operations to transform trees. This approach is extremely powerful, and it is shown to be relationally complete. However, it is hard to implement efficiently [Blake et al. 1992].

Tree Matching [Kilpeläinen and Mannila 1993]: is a query model relying on a single primitive: tree inclusion. The idea is to model both the structure of the database and the query (a pattern on structure) as trees, to find an embedding of the pattern into the database which respects the hierarchical relationships between nodes of the pattern. The language is enriched by Prolog-like variables, which can be used to express requirements on equality between parts of the matched substructure, and to retrieve another part of the matching subtree, not only the root. The complexity of the algorithms is studied, showing that the only case in which the problem is of polynomial complexity is when no logical variables are used and the matches have to satisfy the left-to-right ordering in the nodes of the pattern. Even in the polynomial case, the operations have to traverse the whole database structure to find the answers.

3. A NEW MODEL TO QUERY STRUCTURED DOCUMENTS

We describe now our model. We first expose the main concepts, then the data model and finally the query language.

3.1 Main Concepts

In this section we expose our general ideas on how a structuring model and a query language can be defined to achieve the goals of efficiency and expressiveness simultaneously. Later, we draw the model following these lines.

Our main goal is to define powerful operations that allow matching on the structure of the database, but avoiding algorithms that match "all-against-all", searching across the whole structure tree (e.g. [Kilpeläinen and Mannila 1992]).

A first point is that we want a set-oriented language, because they have been found successful in other areas (such as the relational model), and because if we have to extract the whole set of answers, it is possible to find algorithms that retrieve the elements at a very low cost per processed element.

Since we want to define a fully compositional query language, we consider query expressions as syntax trees, where the nodes represent operations to perform (i.e. *operators*) and the subtrees their operands.

To obtain the set of answers we avoid a "top-down" approach, where the answers are searched in the whole structure tree. We rather prefer a "bottom-up" strategy. The idea is to quickly find a small set of candidates for the answers, and then eliminate those not meeting the search criterion.

Our solution is an algebra over sets of *structure nodes*. These nodes refer to those of the structure tree of the database, each one is a structural element, e.g. a particular chapter or figure. If they cannot be confused with other types of nodes, we refer to structure nodes simply by *nodes*.

The operators take sets of nodes and return a set of nodes. These sets of nodes are subsets of the set of all nodes of the structure tree. The only place in which we pose a text matching query or name a structural component is at the leaves of the query syntax tree. Those leaves must be solved with some sort of index, and converted to a set of structure nodes. Thereafter, all operators deal with those sets of nodes and produce new sets. Figure 1 shows the main concept, which is refined later to detail the query language and to draw a general software architecture for this model.

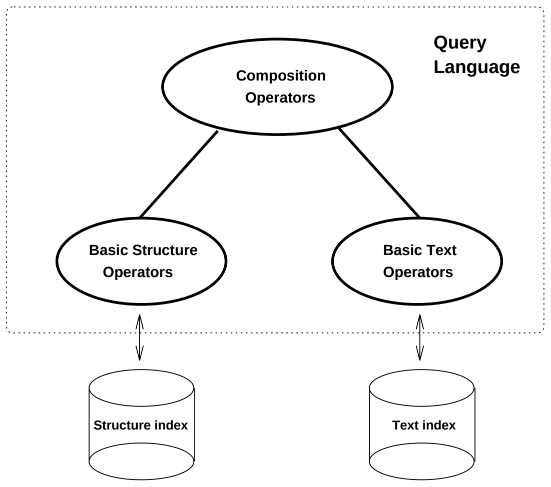

Fig. 1. Initial diagram of how our model operates.

With this approach, we use indices to retrieve the nodes that satisfy a text matching query, or that represent a *node type* (e.g. the chapters). Those nodes must be obtained without traversing the whole database.

Once we have converted the leaves of the query syntax tree into sets of structure nodes, all the other operators take sets of nodes and operate them. Normally, one set will hold the candidates for the result of the operation. Note that we never have to traverse the complete structure when searching.

We need still another piece to complete the picture, since at this point the operations between sets can be as time-consuming as matching against the database.

This piece is the coupling between structure nodes and segments. The segment of a node represents the text it owns, e.g. the segment of a chapter includes all its text. This coupling allows to use efficient data structures to arrange the nodes by looking at their segments (for example, forming a tree). In other approaches [Kilpeläinen and Mannila 1993; Gonnet and Tompa 1987], there is a weak binding between nodes and the segment they own in the text, and thus they need to search in the whole tree to find what they need.

In order for this arrangement to be efficient, the operators should only need to access nodes from both sets that are more or less proximal. When this happens, we can obtain the result by traversing both sets of nodes in synchronization.

If we can efficiently convert text matching expressions and node types into well-arranged sets of nodes, and all operations can efficiently work with the arranged sets and produce arranged sets, then we will have an efficient implementation.

We also show later that many interesting operators are in fact of the kind we need, i.e. they operate on nearby nodes and all what they use is the identity of the nodes and their corresponding segment.

Our point is then twofold: first, we must show that a language in which all operations work on nearby nodes can be efficiently implemented; and second, we must show that it is possible to obtain a quite expressive query language by using only this kind of operations.

This scheme allows to have more than one structure hierarchy, if they are independent. We show later that it also allows integration of other media in a natural way.

3.2 Data Model

We explain now how we model the database. In pass, we redefine more precisely some terms we have been using informally.

A text database is composed of two parts:

—Text, which is seen as a (long) sequence of symbols (characters, words, etc). Whether this text is stored as it is seen, or it is filtered to hide markup or uninteresting components, is not important for the model, since we use the logical view of the text.

—Structure, which is organized as a set of independent hierarchies (i.e. disjoint sets of nodes). Each hierarchy has its own types of nodes, and the areas covered by the nodes of different hierarchies can overlap, although this cannot happen inside the same hierarchy. They do not need to cover the whole text. Again, it is not important for the model how the structure is expressed in or extracted from the text.

Filtering out the markup is important, though. The user should not be aware of details about how the structure of the document is internally represented, or if it is obtained by parsing, etc. He/she should be able to query the document as it is seen in the output device. If two words are contiguous in the logical view, the user should not be aware about that there may be markup between them if, for example, is asking for proximity. It may be argued that including the markup in the text allows the user to query on the markup by text matching. However, we believe that this work must be carried out by the implementation. Any query about markup is probably about structure, and we have a query language for that. The user should not query the structure in such a low-level fashion, he/she should use the content query language to query on content and the structure query language to query on structure.

The text is considered as static, and the structure built on it quite static also. That is, although we allow building new hierarchies, deleting and modifying them, our aim is not to make heavy and continued use of those operations. We are not striving for efficiency in those aspects, our model of usage is: the text is static, the hierarchies are built on it once (or sparingly), and querying is frequent. The way in which the structure is obtained from the text is not part of the model. It can be obtained by parsing the text, by following markup information, etc.

Each *hierarchy* is a tree of *structure nodes* or simply *nodes*, and represents an independent way to see the text (e.g. chapters / sections / paragraphs and pages / lines). The root of each hierarchy is a special node considered to comprise the whole database.

Each hierarchy has a set of *node types* (or "structural components") for its corresponding tree. Examples of node types are page, chapter and section. The sets of node types of different hierarchies are disjoint.

Each node of a hierarchy belongs to a node type of the hierarchy, and has an associated a *segment*, which is a pair of numbers representing a contiguous portion of the underlying text. The segment of a node must include the segments of its children in the tree (this inclusion does not need to be strict).

We point out that this numbering scheme does not need to be noted by a high-level user, but it can be used to represent more logical-oriented constructions, such as collections of documents.

Any set of disjoint segments can be seen as belonging to a special *text hierarchy*, where the nodes belong to a *text* node type. Thus, the text hierarchy has one node for each possible segment of the text. This is an idealized view which never really appears (only disjoint subsets of nodes can be obtained each time, via pattern-matching queries). Observe that there is no hierarchical relationship between any two nodes of such sets. We say that those sets are *flat*.

The disjointness restriction is in fact not essential, since pattern-matching expressions could perfectly well generate a nested structure. However this is not normally the case of text pattern-matching languages.

In the Appendix we give a formal definition of the model, and the query syntax and semantics.

3.3 Query Language

In this section we define a query language to operate on the structure defined previously, including also queries on content.

We do not intend to define a monolithic, comprehensive query language, since the requirements vary greatly for each application. Including all alternatives in a single query language would make it too complex. Instead, we point out a number of operations that follow our lines (and hence can be efficiently implemented).

Each set produced by evaluating a query is a subset of some hierarchy. Those subsets are composed of nodes, not subtrees. Although we normally treat them simply as sets, we can consider that the subsets still keep the hierarchical structure they inherit from the complete hierarchy, therefore forming an ordered forest of trees.

We decided not to merge nodes from different hierarchies in a single result for two reasons: first, it is not clear, hierarchies being different and independent ways to see the same text, whether this could make sense (e.g. pages or chapters with a figure); second, the implementation is much more efficient if every set is a strict hierarchy. In [Clarke et al. 1995], the other choice is selected, i.e. overlaps are allowed in answers, but not nested components.

Although it is not possible to retrieve subtrees (only nodes), the algebra allows to select nodes regarding their "context" in the structure tree (i.e. what is around them), much like in [Kilpeläinen and Mannila 1993].

This language is an operational algebra, not necessarily intended to be accessed by the final user, as the relational algebra is not seen by the users of a relational database. It serves as an intermediate representation of the operations.

3.3.1 *Operations.* We list here the operations we consider sufficient for a large set of applications, and suitable to be efficiently implemented. As we said before, this set is not exclusive nor essential.

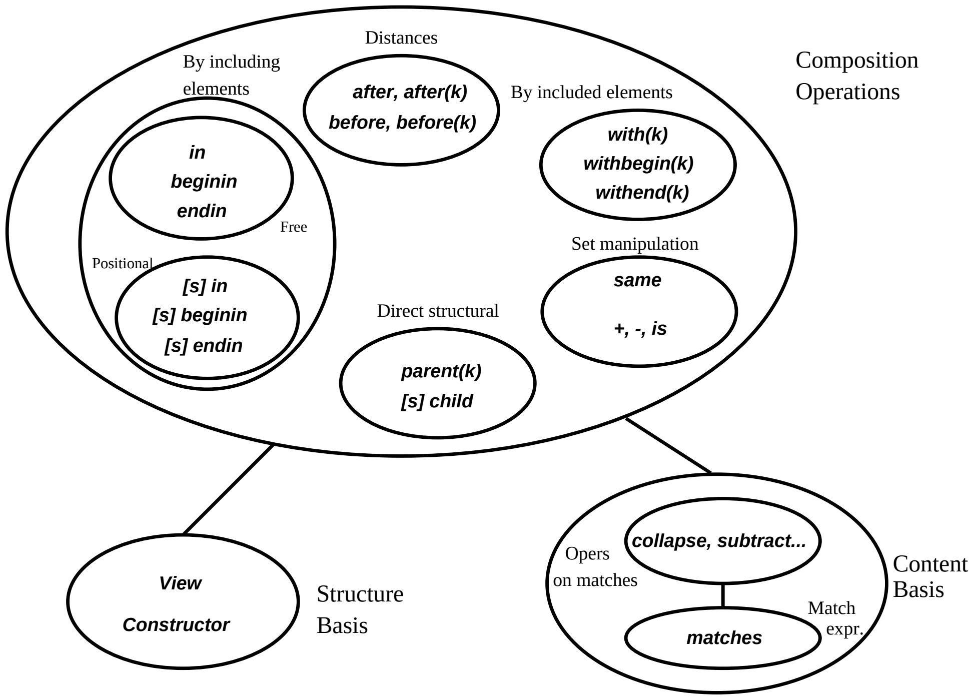

Fig. 2. Possible operations for our model, classified by type.

Figure 2 shows the schema of the 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.

—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 the text hierarchy, as explained before. 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. See [Navarro 1995] for some alternatives.

As an example, we propose M **collapse** M', which superimposes both sets of matches, merging them when an overlap results.

—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. For example, `Formatting` retrieves the whole hierarchy related to formatting aspects. 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.
    —P **beginin/endin** Q: Is the set of nodes of P whose initial/final position is included in a node of Q. For example, `page` **beginin** `formula` are the pages that cut the display of a formula (something that we may want to avoid).
  —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: 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).
    —$[s]$ P **beginin/endin** Q: The same as **beginin/ endin**, but using s as above. For example, `[last] page` **beginin** `chapter` selects the last pages of all chapters (which normally are not wholly included in the chapter).

—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.
  —P **withbegin/withend**(k) Q: Is the set of nodes of P which include at least k start/end points of nodes of Q. If (k) is not present, we assume 1. For example, `chapter` **withbegin**(10) `page` selects chapters with a length of ten or more pages (assuming each chapter begins at a new page).

—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. Both operands must be from the same hierarchy, which cannot be the text 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. Except for **same**, both operands must be from the same hierarchy (which must not be the text hierarchy).

—$P + Q$: 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. P and Q can be from different hierarchies. For example, `title` **same** `"Introduction"` gets the titles that say (exactly) "Introduction".

Observe that all operations related with beginnings and endings make sense only if the operands are from different hierarchies, since otherwise they are the same as their full-segment counterparts.

3.3.2 *Examples.* We present some examples of the use of the language, to give an idea of what kind of queries can be posed.

Suppose we have a hierarchy V with main structural component `book`. A `book` has an `introduction`, a number of `chapters`, a `bibliography` and an `appendix`. Each of them may have `sections` (which may have more `sections` inside), as well as `formulas`, `figures` and `tables`. A `table` is divided in `rows`, and these in `columns`. A number of `paragraphs` may appear in `chapters`, `sections`, `introduction` and the `appendix`. The following elements have always a `title`: `book`, `chapter`, `section`, `figure` and `table`. Finally, we have `citations` which references other books, listed under `bibliography`.

We have another hierarchy V' with `volume`, `page` and `line`, and a hierarchy VP for presentation aspects, e.g. `italics`, `emphasize`, etc.

Suppose also that we have a simple word matching language for text.

—`chapter` **parent** `(title` **same** `"Architecture")`, is the set of all chapters of all books titled “Architecture”. Here, `"Architecture"` is an expression of the pattern-matching sublanguage.

—`[last] figure` **in** `(chapter` **with** `(section` **with** `(title` **with** `"early")))`, is the last figure of chapters in which some section (or subsection, use **parent** to select top-level sections) has a title which includes the word “early”. This query is illustrated in Figure 3.

—`paragraph` **before** `(paragraph` **with** `("Computer"` **before** `(10) "Science" (paragraph))) (page)`, is the paragraph preceding another paragraph where the word “Computer” appears before (at 10 symbols or less) the word “Science”. Both paragraphs must be in the same page.

—`[3] column` **in** `([2] row` **in** `(table` **with** `(title` **same** `"Results")))`, extracts the text in position $(2, 3)$ of tables titled “Results”.

—`(citations` **in** `([2..4] chapter` **in** `book))` **with** `"Knu*"`, selects references to Knuth's books in chapters 2-4.

—`(section` **with** `formula)−(section` **in** `appendix)`, selects sections with mathematical formulas that are not appendices.

—`introduction` + `(chapter` **parent** `(title` **with** `"Conclusions"))` + `bibliography`, can be a good abstract of books.

4. EXPRESSIVENESS

We first compare formally our model against the other similar models surveyed (except *p-strings*, since it is a structure manipulation model). Although this point-to-point comparison is useful, no formal categorization of expressiveness features exists. Therefore, we appeal to informal methods. We show that our model fits the quality criteria exposed in [Sacks-Davis et al. 1994]. We also develop an informal framework to situate models of this kind.

4.1 A Formal Comparison

In the Appendix we formally define the semantics of our operations. That definition is used to compare our model against each of the novel models, to determine which

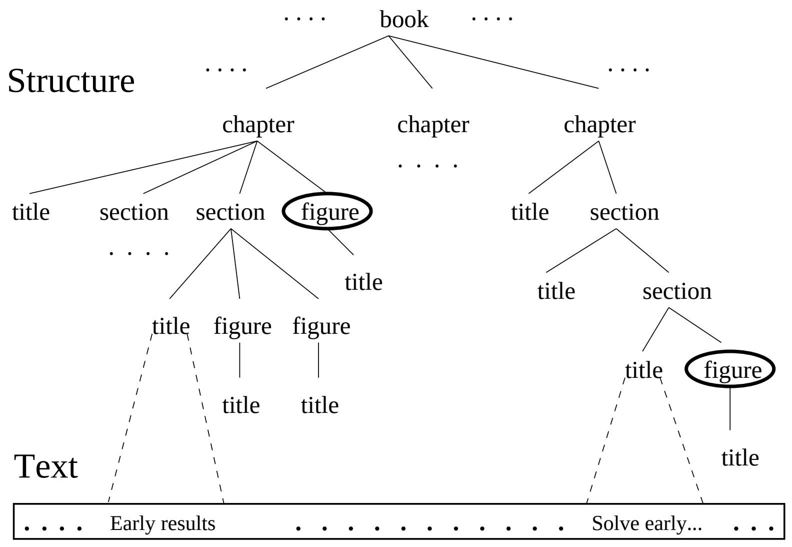

Fig. 3. Illustration of the effect of the query `[last] figure` **in** (`chapter` **with** (`section` **with** (`title` **with** `"early"`))). The circles indicate selected nodes.

features from ours can be represented in others and vice versa. A brief abstract of the results follows, omitting the representation mapping performed between the models (see [Navarro 1995] for details).

The Hybrid Model [Baeza-Yates 1996]: our model can completely represent it, while the converse is weak. Although it can represent a structure defined in our model, little can be queried about it (e.g. ancestorship).

PAT Expressions [Salminen and Tompa 1992]: our model can almost completely represent it, disregarding some undesirable complications of the language that one really would not want to represent (mainly regarding conversions between regions and match points). The converse is again weak, since this model cannot represent recursive structures, which prevents it from representing almost all the operations of our language.

Overlapped Lists [Clarke et al. 1995]: the comparison is difficult because the models are almost orthogonal. Ours can represent hierarchies but not overlaps, while this one does the inverse. Disregarding the fact that each model cannot represent the most important structure of the other one, most operations can be translated between both models.

Lists of References [MacLeod 1991]: our model can completely represent this model (recall that we consider the part of the model related to querying structures). On the other hand, a good portion of our model can be represented in this one, being the most important omissions the unability to have multiple hierarchies and to return nested components.

Tree Matching [Kilpeläinen and Mannila 1993]: we can represent part of this model. The most important omissions are of course logical variables (which make the model implementation NP-complete), and the inclusion semantics. These are different in both models: in this one, an inclusion relationship must hold in the text if and only if it holds in the query, while in ours it must hold in the text if it holds in the query (but more relations can hold in the text). This prevents each model representing the other in this aspect, although we show that we can represent some restricted cases.

This is related to what in [Consens and Milo 1995] is called the *both-included* problem, namely the ability to express "a containing (b followed by c)". In [Consens and Milo 1995], a simplification of PAT expressions is used to formally analyze its expressive power, finding that *both-included* cannot be represented without introducing tuples and join capabilities into the language (ala relational). The same holds for our model. Observe that the source of this problem is that it is not possible to express that a name appearing in two parts of an expression should denote the same node, and that is exactly what a relational equijoin would do.

By using logical variables, this model can represent almost all of ours, being its weak part the integration between text and structure.

4.2 Quality Criteria

In [Sacks-Davis et al. 1994], a number of queries that this kind of language should be able to answer are pointed out. We summarize them here to show that we can express all in the areas we are interested in (i.e. we exclude the features related to relevance ranking and connection to relational databases, which we do not address in this work).

—Word-by-word access, e.g. "find ⟨*doc*⟩s containing 'parallel' and ('computing' or 'processing')" can be expressed as (`doc` **with** "parallel") **with** ("computing" **collapse** "processing").

—Query scope restricted to sub-documents, e.g. "find ⟨*doc*⟩s with ⟨*title*⟩ containing 'parallel' and 'processing' " can be expressed as `doc` **parent** ((`title` **with** "parallel") **with** "processing"). The other example in the paper is "find ⟨*doc*⟩s with 1st ⟨*para*⟩ containing 'parallel' and 'processing' ", that can be expressed as `doc` **with** ((([1] `para` **in** `doc`) **with** "parallel") **with** "processing").

—Retrieval of sub-documents, e.g. "find ⟨*section*⟩s with ⟨*para*⟩s containing 'parallel' and 'processing' " can be expressed as `section` **with** ((`para` **with** "parallel") **with** "processing").

—Access by structure of documents. Many examples are presented here:

 —"Find elements with parent of type ⟨*article*⟩" can be expressed as `All` **child** `article`, where `All` is the name of the hierarchy.

 —"Find elements with children" can be expressed as `All` **parent** `All`.

 —"Find elements where the first child is ⟨*title*⟩" can be expressed as `All` **parent** ([1] `title` **child** `All`).

 —"Find elements within a ⟨*section*⟩" can be expressed as `All` **in** `section`.

 —"Find ⟨*doc*⟩s that contain a ⟨*corres*⟩ can be expressed as `doc` **with** `corres`.

 —"Find ⟨*section*⟩s that contain a ⟨*section*⟩" can be expressed as `section` **with** `section`.

—Access to different types of document, e.g. "Find articles, papers and books with 'parallel' and 'computing' in the title" can be expressed as (`article` + `paper`

Model	Type of structure	Implicit or explicit	Static or dynamic	Bound to	Answers
Ours	Hierarchy	Explicit (multiple)	Static	Intermediate	Nested
Hybrid Model	Flat	Implicit	Static	Text	Flat
PAT Expressions	Hierarchy (not recursive)	Implicit	Dynamic	Text	Flat
Overlapped Lists	Hierarchy (with overlaps)	Implicit	Dynamic	Text	Overlapped
Lists of References	Hierarchy (and network)	Explicit (single)	Static	Structure	Flat (and of the same type)
Tree Matching	Hierarchy	Explicit (single)	Static	Structure	Nested
p-strings	Hierarchy	Explicit (multiple)	Dynamic	Intermediate	Nested

Table 1. Analysis of structuring power.

+ `book`) **with** ((`title` **with** "parallel") **with** "computer"). This issue is more concerned with the problem of having the different names standing for "title" in each type of document, but this is also easily handled: (`book` **with** `booktitle` ...) + (`article` **with** `articletitle`...) + ...

—Access by attributes, e.g. "find $\langle corres \rangle$s with attribute 'confidential' = yes". If we have those attributes as node types children of the node and their values in the text, we can answer simple queries, in this case we express it as `corres` **parent** (`confidential` **same** "yes").

4.3 An Informal Framework

We use the experience gained in the formal comparison to define an informal framework to situate the models regarding their important features [Baeza-Yates and Navarro 1996; Navarro and Baeza-Yates 1995a]. This framework divides the analysis into two main areas:

—Structuring power (i.e. how the database is structured). See Table 1.
  —Type of structure (flat, hierarchical or network)
  —Implicit or explicit structure (i.e. using embedded markup or an explicit structure index)
  —Static or dynamic structure (i.e. ability to reindex)
  —Link between content and structure (strongly text-bound, intermediate, or strongly structure-bound)
  —Structure of answers (flat, overlapped or nested)

—Query language (i.e. what can be asked). See Table 2.
  —Text matching (i.e. querying content)
  —Set manipulation (i.e. handling sets of answers)
  —Inclusion relationships (i.e. selecting nodes included in or including others)
  —Distances (i.e. selecting nodes at a given distance to others)

There is a third area regarding how the content is seen, but we do not consider that part in this work.

Model	Set manipulation	Inclusion relationships	Distances
Ours	Yes (same hierarchy). A different set for nodes and for text	Including n and included. Direct and positional inclusion	Both distance-bound and minimal. Inside a given node
Hybrid Model	Separate for text and documents. Complement	Restricted to fields	Only in matches
PAT Expressions	Yes. Also negation of operations	Including n and included	Yes, distance-bound
Overlapped Lists	Union and combination	Including and included, plus negations	Combination and "n words"
Lists of References	Yes, but only for nodes of the same type	Including n and included. Restricted direct	None
Tree Matching	Yes, via logical connectives	Tree patterns + variables	None

Table 2. Analysis of query languages.

Figure 4 presents a simplified graphical version of this comparison. We identify the main points about expressiveness, and represent each model as a set containing the aspects it reasonably supports.

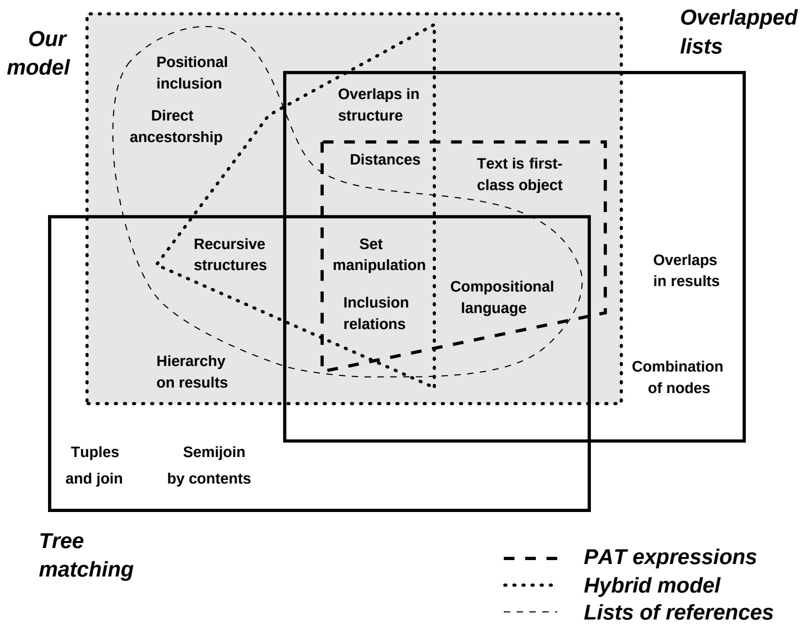

Fig. 4. A graphical representation of the expressiveness comparison.

From the figure, we can see that the main features our model lacks are tuples, semijoin by content (i.e. retrieve all chapters whose titles appear in this paragraph) and the possibility of having overlaps and combine nodes in the result set of a query.

We believe that none of them can be included without significantly degrading the performance. The features we support are enough for a large class of applications. Some of the lacking features are better included by integrating this model with another one.

5. IMPLEMENTATION ISSUES

In this section we cover the main aspects related to the implementation of the model, regarding indexing and querying. Later we depict a suitable software architecture and a prototype we implemented to test the model. More details can be found in [Navarro 1995].

We represent each hierarchy as a tree of nodes. Those trees form our structure index. From that index, the **Struct** and **Hierarchy** queries obtain other types of trees (rootless) that represent sets of nodes (the algebra on which the rest of the language operates). The same does the processing of content retrieval, by using its own index. As well as in the model, the implementation of the content retrieval is independent of the rest and is not studied here.

Thus, there are two very different operations: the leaves of the query syntax tree must be solved with an index, while the internal nodes operate and produce sets of nodes. We first explain how to process the internal nodes of the query syntax tree, then the leaves (this includes the indexing scheme) and then the whole evaluation plan. Finally, we formally analyze the efficiency of this implementation.

It is important to observe that we are not proposing any new implementation technique. For example, those can be found in [Mackie and Zobel 1992]. Rather, we use well-known techniques to efficiently implement the model. The key of the efficiency achieved is in the definition of the model, which allows an efficient implementation with classical tools.

5.1 Evaluating the Internal Query Nodes: Traversal Algorithms

Since the language is compositional, all the operations except **Struct** and **Hierarchy** receive and deliver sets of nodes. These sets are also arranged into trees, attending to their ancestorship in the corresponding hierarchy (text queries return flat sets, implemented as rootless trees with all their nodes in the first level).

Since only proximal nodes are related in the operations, all the algorithms traverse both trees in synchronization. The idea is much like list merging, but in this case each node has a next sibling and a list of children. Each operation traverses the trees in a slightly different way and performs slightly different operations, but the central idea is the same.

Two implementations are possible: a full-evaluation scheme computes the whole set of answers at once; while a lazy-evaluation scheme computes only the result, and nodes from inner operands of the query syntax tree are obtained only if they are necessary to compute the final result.

The lazy mechanism works as follows: from the syntax tree of the query we require the first level of answers. This triggers new requirements to the children of the root of the syntax tree, which in turn expand the first level of their answers, and so on.

Hopefully, not all the sets involved in the expression need to be fully evaluated. This mechanism is not new, for example is of widespread use in lazy functional languages and object-oriented query languages.

The lazy version forces an order of evaluation that is not always optimal, since it is given by the requirements of operators higher in the query syntax tree. Because of this, it has higher complexity. Since, on the other hand, it may compute only part of the result, it is not immediate which one is better. Experimental results (see later) show that lazy evaluation is normally better. Lazy evaluation is also suitable for interactive environments in which the user wants to see a top-level answer and then navigates only into some subtrees, thus avoiding the need to evaluate the rest.

5.2 Evaluating the Leaves of the Query Tree: Indexing Schemes

What is left to consider is how to efficiently solve **Struct** and **Hierarchy** queries. These are handled by accessing an index. The requirements for this index are

(a) Given a structural component id, retrieve the tree of all nodes of that type.

(b) Given a hierarchy, retrieve the tree of all its nodes.

For lazy evaluation, instead, we keep a pointer to a node in the disk, and ask to

(c) Given a node, retrieve all its top-level descendents of the same type.

(d) Given a node, retrieve all its children

Those operations must be preferably linear (in the size of the answer), counting both CPU processing, number of disk accesses, and total seek time. Observe that the total seek time may be of higher order than number of disk accesses, for example $O(n)$ random accesses to a file take $O(n^2)$ seek time on average.

There are many alternatives to handle the trees of the structure index on disk. Each one leads to a different indexing scheme.

5.2.1 *A Full Index.* A full index stores the hierarchy tree in a breath-first layout on disk. For each node, enough pointers are kept to perform (a) and (b) in linear expected time with respect to the size of the output (although the seek time for (a) is proportional to the size of the whole hierarchy).

The index is split in one file per level of the tree, to ease the reindexing process. With the same purpose, the endpoints of the segments of each node are computed relative to their parent segment. This does not pose any implementation problem, since a node is only accessed via its parent. The benefit is that complete subtrees can be inserted or deleted without modifying the numbers in the rest of the index. When the results of **Struct** and **Hierarchy** queries are extracted from the index, the positions are rewritten to be absolute.

Although in most cases it suffices to retrieve the node id and the segment of each node in order to further process the query, sometimes the node id of the parent is also needed (for **parent** and **child** queries). It is possible to syntactically determine from the query when this is needed, to avoid any further access to the disk.

If there is enough memory, it is more efficient to read, in a single pass, all the **Struct** nodes needed for all the leaves of the query syntax tree.

Observe that this index is not very well suited for lazy evaluation, since it cannot efficiently implement (c). This is because the top-level descendents of a node of the same type can be spread across the file, even unordered, so the seek time may be large. Another drawback of this index is its large space requirement (7 words per node).

5.2.2 *A Partial Index.* It is possible to reduce the space requirement to just 2 words per node, at the cost of not allowing **parent/child** queries, nor two different nodes having the same segment (this last restriction is easily overcome at no expressiveness cost).

In this case, two files are kept for each node type, one with the sorted start points and the other with the sorted end points. See Figure 5.

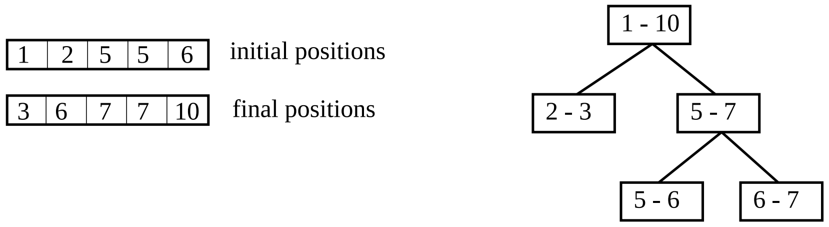

Fig. 5. An example of a partial index and its associated hierarchy.

It is easy to implement (a) in linear time and in a single pass over the index, but for (b), the **Hierarchy** query must be solved as the union of its node types (this takes $\approx kn/2$ time, being n the size of the hierarchy and k the number of different node types). Finally, the index must be wholly rewritten upon reindexing, and this scheme is worse than the previous one for (c).

5.2.3 *An Index Suitable for Lazy Evaluation.* With 3 words per node, a version of the partial index suitable for lazy evaluation can be designed. This can be extended to a full index of 4 words per node (by adding the parent of each node), that, however, still cannot handle different nodes with the same segment.

In this case, a file for each node type is kept, with a layout similar to the full index. This allows to implement (a) and (c) in linear time, but (b) and (d) must be solved as before. This is not a big problem with lazy evaluation, since most probably the whole hierarchy is never computed. The problem of different nodes having the same segment can be solved by the parser, with artificial intermediate points.

5.3 The Whole Query Evaluation: Query Plan

Given a query syntax tree, any evaluation order that respects the dependencies is allowed. We prefer the one that minimizes the total amount of memory needed for the whole evaluation. The sequence of operations to perform to solve the query is called a *query plan.*

The problem of generating minimum-memory query plans is solved, although simplified, in [Aho et al. 1986]: one has to obtain first the larger operand, then the other operand, and then operate them. Since in [Aho et al. 1986] the problem is using registers to compile an expression tree, the weight of a tree is defined as the number of its nodes. We should instead estimate the size of our sets. In absence of good estimators, using the number of nodes seems a reasonable choice. Thus the algorithm would be as simple as solving the tree by evaluating the bigger subtree first.

Operation	Full	Lazy
+,−	n	$n \min(d, h)$
is/same	n	n
in	$\min(n, d^2 h)$	$\min(n, d^2 h)$
beginin/endin	$\min(n, d^2 h)$	$\min(n + dh, d^2 h)$
$[s]$***in**	$n \min(d, h)$	$n \min(d, h)$
with*(k)	$k = 1 \; ? \; n \; : \; nh$	$n \min(n, k + dh)$
$[s]$**child**	n	n
parent(k)	n	$n \min(d, h)$
after/before(C)	$C = \emptyset \; ? \; n \; : n \min(n, dh)$	

Table 3. Time complexities of the algorithms (n is the size of the operands, h the maximum height of their tree representation and d the maximum arity of those trees.)

A useful alternative to a syntax tree to represent the query is to have a directed acyclic graph (DAG), to avoid re-evaluating common subexpressions. In that case, the problem of finding an optimal evaluation order becomes much more complicated, being similar to the problem of evaluating the DAG of an expression minimizing registers, which is known to be NP-Complete [Garey and Johnson 1979]. Some heuristics for this case are presented in [Aho et al. 1986].

These policies to evaluate a query in a given order are only applicable to full evaluation. Lazy evaluation expands the nodes in an unpredictable way, which for each node of the syntax tree is dictated by the requirements posed by its parent.

Another important point is that we can write our algorithms to operate by modifying one of the operands to produce the answer, or to generate a new set. If the operand is to be used only once, it is better to modify it, otherwise we should generate a new set. The query plan generator must implement the appropriate policy to avoid keeping unnecessary copies in memory, deleting or replacing operands the last time they are used.

Another interesting point is the optimization of the query, but we do not address that issue here, since it is complex enough to constitute a whole separate problem.

If, despite the clever algorithms to avoid it, the operands are too large to fit in memory, a swapping policy must be implemented. The problem can be solved by a virtual-memory-like approach, keeping part of the intermediate results swapped out to disk. In this case, we must select the operand which will be used later to swap it out (that information is available from the query plan). An interesting option to store those internal results is to use the same layout as the one we use for the indices of node types. This idea together with a good swapping policy provides a uniform and elegant solution to the problem.

5.4 Analysis

We analyzed the efficiency of our algorithms both theoretically and experimentally. Table 3 abstracts the complexity analysis. Observe that the lazy version, as expected, has higher complexity than the full one. See [Navarro 1995] for more algorithmic details. The performance of **Hierarchy** and **Struct** queries depend on the indexing scheme, and was explained in Section 5.2.

6. A SOFTWARE ARCHITECTURE

In this section we outline a possible software architecture for a system based on our model.

Users should interact with the system via an interface, in which they define what they want in a friendly language (e.g. [Kuikka and Salminen 1995; Kilpeläinen and Mannila 1993]). That interface should convert that query into a query syntax tree, i.e. the language we present here. This tree is then submitted to the query engine.

The query engine optimizes the query and generates a smart query plan to evaluate it (i.e. converts the tree into a sequence of operations to perform). The leaves of the query tree involve extracting components of the hierarchy by name (node types), and text matching subexpressions. The first ones are solved by accessing the index on structure to extract the whole set of nodes of that type (i.e. a set of node *id*s and their segments). The second ones are submitted to the text search engine, which returns a list of segments corresponding to matched portions of the text. Thereafter, the rest of the operations are performed internally, until the final result (a set of nodes) is delivered back to the interface.

The interface is in charge of visualizing the results. To accomplish that, it must access the contents of the database, at the portions given by the retrieved segments. This is also done via a request to the text engine, since only that engine knows how to access the text.

The text engine is in charge of offering a text pattern-matching language, keeping the indices it needs for searching, and presenting a filtered version of the text file to upper layers. The first service accepts a query and returns a list of matching segments. The third one accepts a segment and retrieves its text contents.

If the text engine is a completely separate subsystem, two separate indexing processes may exist. One of them indexes the text to answer text pattern-matching queries (this indexing is performed by the text engine). The other extracts the structure in some way from the text (parsing, recognizing markup, etc.), and creates the structure index, which is later accessed by the query engine. This is the only time when the text can be accessed directly from outside the text engine.

Indeed, both indexers must collaborate, since the markup used by the structure indexer should be filtered out by the text indexer when presenting the text to upper layers.

See Figures 6 and 7 for a diagram of how a complete system based on this scheme could be organized. The “document layer” is intended to support more sophisticated document management, such as collections of documents.

7. EXPERIMENTAL RESULTS

We implemented a prototype following the proposed software architecture, to test average time and space measures, as well as to evaluate heuristics. Our aim is not to compare our model against other systems, which is very difficult in practice, but to test the suitability of our model for implementation.

For the matching sublanguage, we use the API of SearchCity [Ars Innovandi 1992], which is based on the use of PAT or suffix arrays [Manber and Myers 1990; Frakes and Baeza-Yates 1992]. The matching sublanguage supported by this API includes: whole words, ranges, wildcards, proximity search, boolean operators, sub-

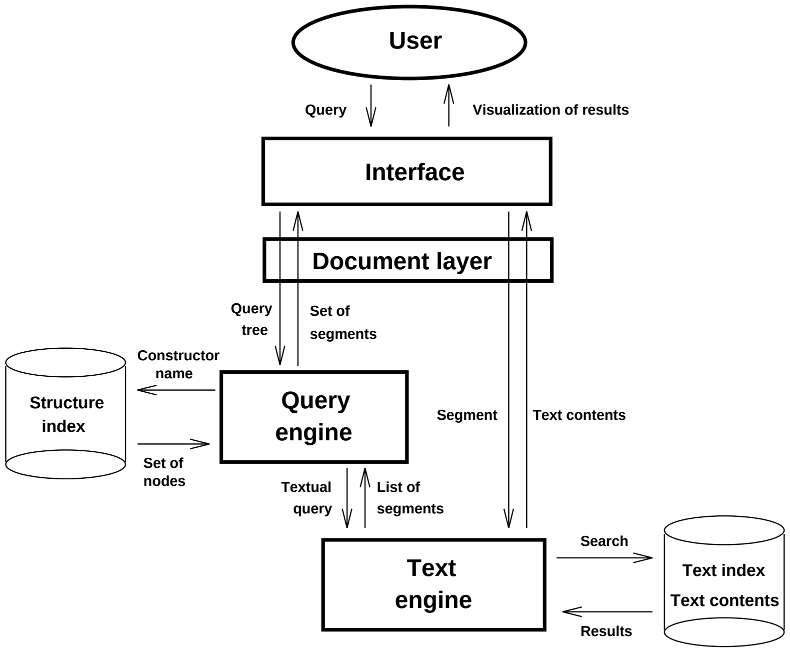

Fig. 6. The architecture of a system following our model, regarding querying.

traction and fuzzy search. The API allows filtering of the raw text, by applying a format filter, character normalization filter, and a synonyms and stopwords filter. The first filter allows to use files in other formats different from ASCII, without copying their filtered form into a new file. All format-related (i.e. non-searchable) portions of the file are filtered out, so that queries can only see the true contents of the file.

We use that filtering facility to incorporate texts whose structure is embedded in their content, this way allowing only the contents to be searchable, and using the marking to parse the structure and generate a structure index. Hence, we have an index for matches and a separate index per hierarchy. The PAT array can be seen as a generic index for the text hierarchy.

We have implemented filters for DDIF [Digital Equipment Corporation 1991], SGML [International Standards Organization 1986], LaTeX[Lamport 1986] and C code [Kernighan and Ritchie 1978]. The query plan generation is still very simple, no query optimization is performed.

Currently, the structure index is kept in memory, although in the future it will be kept on disk. Both the text and the text index are kept on disk. This is not a serious limit as it could appear, since the structure index tends to be far smaller than the text. For example, one megabyte of texts and articles generates near 1000 nodes, i.e. an index of 28 kilobytes using the most space-demanding indexing scheme. With only 8 megabytes of RAM devoted to the index we can handle a database of 300 megabytes of text. There are of course other types of documents that pose more serious problems. For example, one megabyte of C code indexed via

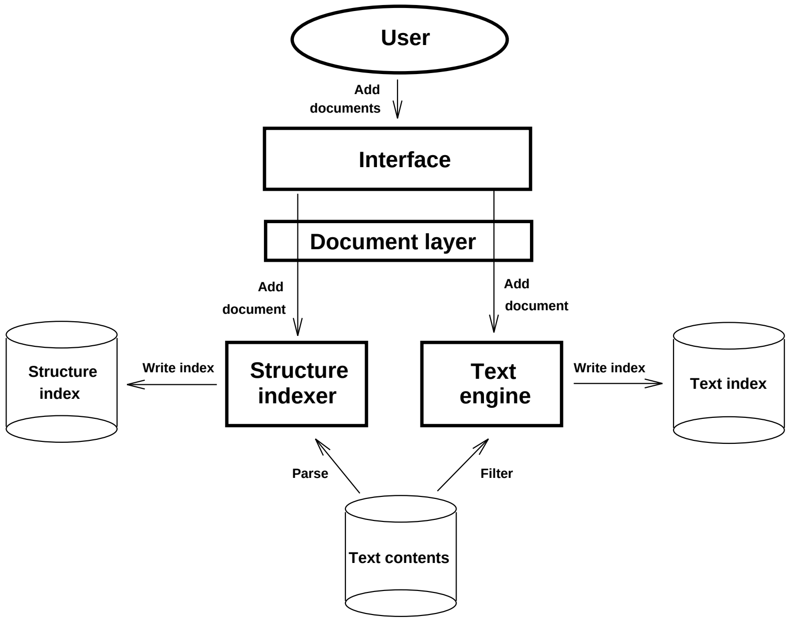

Fig. 7. The architecture of a system following our model, regarding indexing.

a fine-grain parsing can generate an index of 300,000 nodes, i.e. up to 10 megabytes for the full index.

We conducted a set of tests running our prototype implementation on a Sun SparcClassic, with 16 Mb of RAM, running SunOS 4.1.3_U1. The CPU speed of this machine is approximately SpecMark 26.

From these results we conclude that, in the full version, the time to process a query is proportional to the total number of nodes of all internal results, being the constant near 50,000 nodes per second for our machine. A rough approximation is $(2q - 1) \times (\textit{average operand size})$, where q is the number of nodes of the query syntax tree. The lazy version is normally better than the full one, especially for complex queries, although its running times are very unstable. The running times are between 25% and 90% of the full version, and between 40% and 100% of the nodes are computed. Figure 8 shows typical times for a single operation.

These good results are possible thanks to our approach of operating proximal nodes, which allows to compute the results by a one-pass traversal through the operands. The ideas of a set-oriented query language, a data structure to efficiently arrange segments, and the reduction of all queries to operations on proximal nodes lead to an implementation where the amortized cost per processed element is, in most cases, constant.

8. EXTENDING THE MODEL TO MULTIMEDIA DOCUMENTS

We study in this section how to extend the model to handle other types of data. We are interested in documents containing not only text, but also audio, images, video, etc. being queried [Subrahmanian and Jajodia 1996].

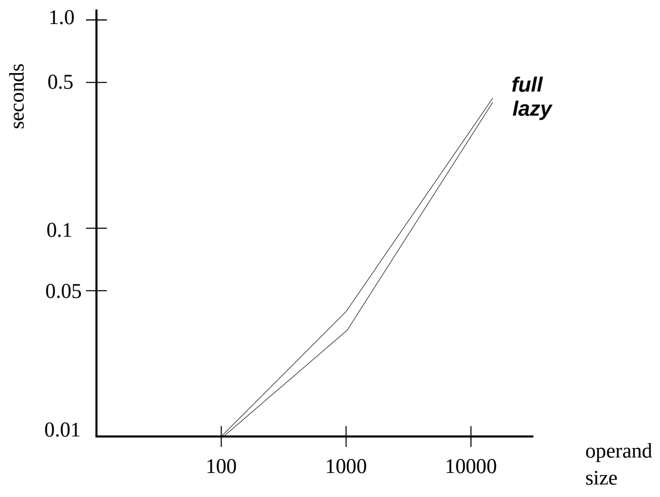

Fig. 8. Typical times for a single operation query. Note that the scale is logarithmic in both axes.

Multimedia documents have normally a structure that relates their components, be them text, images, etc. We call that structure the "external structure". However, not only the text but also the multimedia objects can have internal structure. For example, it is possible to structure an audio segment corresponding to a symphony.

Although text has been normally queried by content, in classical multimedia databases multimedia objects were not queried truly by content [Bertino et al. 1988]. Rather, a textual or formatted data description (which is created by hand) was attached to each object, and that was the only information queries could ask on them. Multimedia data was in this way introduced into the classical formatted data model, leaving unresolved only performance and security considerations [Elmasri and Navathe 1994].

Recently, advances in signal processing and artificial intelligence techniques have allowed the possibility of searching by content into objects such as audio and images (e.g. [Christodulakis and Faloutsos 1986; Wu et al. 1994]). Our purpose is to show how those capabilities can be included in a data model to smoothly integrate the new facilities into existing capabilities. Moreover, we show how the structure inherent to multimedia documents, both external to the multimedia objects and internal to each one, can be exploited to improve the retrieval capabilities of a database.

For our fundamental ideas to remain applicable, we assume that it is still possible to impose a hierarchical structuring on the data. This assumption is quite general, although simpler than hypermedia or spatial structuring. Hypermedia structuring could be supported by differentiating between "structure" links (which are hierarchical) and "pointer" links (which are to be queried by other means, e.g. a Graphlog-like language [Consens and Mendelzon 1993].

For technical reasons (i.e. to represent structure by containment of segments), we assume that the different elements inside each structure can be linearly ordered.

This is done internally and it is *not* visible to the user. Queries about distances are not permitted to spread among different type of objects (we return to this later). If the document does not have order between elements, positional queries can be simply not permitted, hiding the (artificial) ordering to the user.

Thus, our database is now composed of

—Data: a sequence of segments of heterogeneous types.

—Structure: as before.

Each relevant position of the data stream is assigned a number. Those numbers are in ascending order, but they no longer represent consecutive text symbols. They just indicate the ordering (and containment) between the elements. Because of this, our structural **after/before** operation no longer makes sense, and must be replaced by a medium-specific **after/before** where applicable (e.g. positions may represent words in the text and seconds in the audio). Distances make sense only inside a homogeneous data portion or containing complete data portions, where the interpretation of their start/end points is specific to that medium.

Instead of a single sublanguage for (text) content, we have such a language for each medium. For example, the language for audio contents may consist of audio pattern-matching (by signal processing), the one for images may be oriented to (precomputed) semantic descriptions. The answers are regarded as segments from a specific hierarchy (one for each medium, not only the text hierarchy).

Although each medium interprets the structure in its own way, we must keep the consistent idea of containment. For example, it may represent a part-of relation on the objects appearing in an image, each node representing an object with an associated "segment". Other queries (e.g. spatial proximity) are to be regarded as medium-specific.

This technique allows to consider the structure of a document as homogeneous, regardless of whether it is external or internal. All the queries about containment, etc. performed on the structure extend seamlessly to the structure defined inside images, audio, etc.

From the software point of view, there is a module responsible for handling each type of data. A general parser must recognize the external structure and each data object, passing the relevant segment to the corresponding indexer, who indexes its content and internal structure. An index for each medium is built with the content, while the internal structure is returned to the general indexer, that appends it to the hierarchy. At the moment of querying, subexpressions belonging to the content-retrieval sublanguage of each medium is assigned to a specific query processor, which uses its own index (be it for text retrieval, audio matching, image semantics, etc.) to retrieve a set of segments that represents the relevant portion of the object. The structural part of the query is solved as before, regardless of its external or internal nature.

To visualize the final result, the interface sends the segment to a module that determines which medium it belongs to, and passes the request to the appropriate specific module. This may end in the retrieval of a text portion, an image, a combination of text and audio, etc.

Figure 9 shows an example document. A query such as `Recital` **`in`** `(Phrase` **`with`** `(Photo` **`with`** `(Paul near me)))` retrieves the segment 10–12, visualized as the sound recording of the recital.

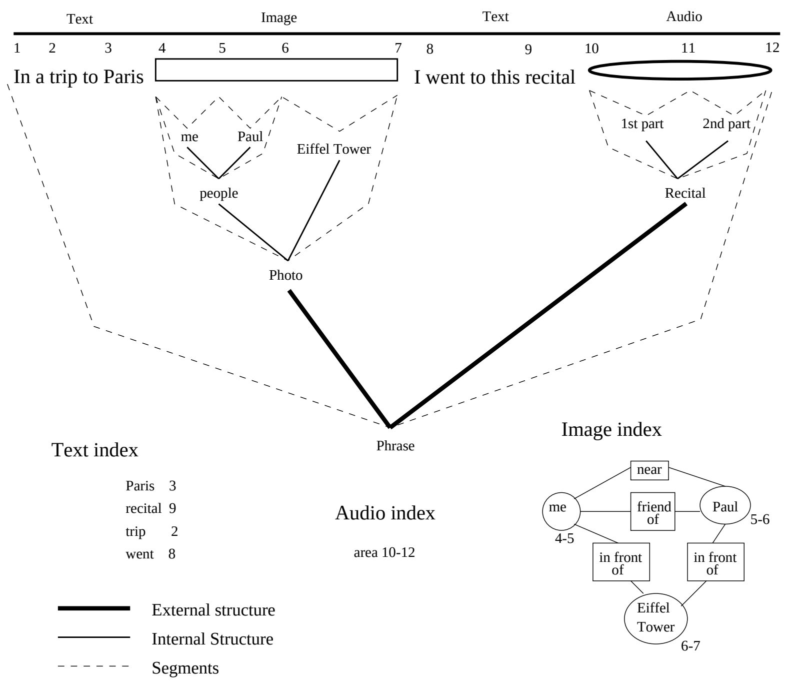

Fig. 9. A document with multimedia elements, with its structure and contents indices.

9. CONCLUSIONS AND FUTURE WORK

The problem of querying a document database on both its content and structure has been analyzed. We found the existing approaches to be either not expressive enough or inefficient.

Hence, we have defined a model for structuring and querying textual databases that is expressive enough and efficiently implementable, and extended it to handle other types of data. The language is not meant to be accessed by final users, but to constitute the operational algebra.

We have evaluated our model in expressiveness and efficiency. We showed it to be competitive in expressiveness, getting close to others that do not have an efficient implementation. On the other hand the algorithms show good performance, both in their analysis and the experimental tests. That situates this model close in efficiency to those much less expressive.

See Figure 10 for a graphical (and informal) comparison of similar models when taking into account both efficiency and expressiveness. Note that we have included *p-strings* in this drawing, assuming an expressiveness superior to all the languages we have analyzed. Note also that only a part of the lists-of-references model is considered (and the efficiency to implement only that part is considered). Observe that, as any quantization of concepts, this comparison is subjective. Nevertheless, it does give an idea of where our model lies.

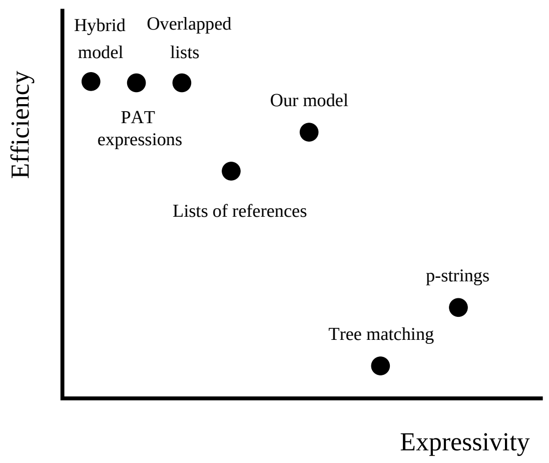

Fig. 10. A comparison between similar models, regarding both efficiency and expressiveness.

There are a number of research directions related with this work:

—Further exploration of the possibilities offered by the model in order to find more operators which fit into our philosophy (being thus efficiently implementable).

—Definition of a query language suitable for end users, possibly visual, to map onto our operational algebra (see for example [Kuikka and Salminen 1995]).

—Integration between this kind of model and others, such as the relational model or the traditional ones of information retrieval. This issue has not been considered here, since we focus on the structure problem. An important issue is how to include relevance ranking in our model (see [Sacks-Davis et al. 1994] and [Arnold-Moore et al. 1995] for some ideas).

—Generalization of the problem to manage non-hierarchical structures, such as a hypertext network, while keeping the desirable properties obtained for this simpler case. A recent work [Dao et al. 1996] that extends Overlapped Lists to handle nesting and overlapping at the same time shows a different trend that deserves attention.

—A formal framework in which to compare expressiveness is needed. The long-term goal is a formal and sound hierarchy like what can be found in the area of formal languages (see [Navarro and Baeza-Yates 1995a; Consens and Milo 1995] for some examples).

—There are a number of implementation issues remaining, for example to design a good swapping policy for query evaluation, keeping the index on disk, handling multimedia documents, query optimization, etc.

ACKNOWLEDGMENTS

We thank Patricio Poblete, Nivio Ziviani, Mariano Consens, José Blakeley, Omar Alonso, and the unknown referees for their fruitful comments and careful reading that improved this work.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques and Tools.* Addison-Wesley.

Arnold-Moore, T., Fuller, M., Lowe, B., Thom, J., and Wilkinson, R. 1995. The ELF data model and SGQL query language for structured document databases. In *Proc. of the 6th Australasian Database Conference* (1995), pp. 17–26.

Ars Innovandi. 1992. *Search City 1.1. Text Retrieval for Windows Power Users.* Santiago, Chile: Ars Innovandi.

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

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

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

Bertino, E., Rabitti, F., and Gibbs, S. 1988. Query processing in a multimedia document system. *ACM TOIS 6*, 1 (Jan.), 1–41.

Blake, G., Bray, T., and Tompa, F. 1992. Shortening the OED: Experience with a grammar-defined database. *ACM TOIS 10*, 3 (July), 213–232.

Cattell, R. 1991. *Object Data Management.* Addison-Wesley.

Christodulakis, S. and Faloutsos, C. 1986. Design and performance considerations for an optical disk-based multimedia object server. *IEEE Computer 19*, 12 (Dec.).

Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. 1994. From structured documents to novel query facilities. In *Proc. ACM SIGMOD'94* (1994), pp. 313–324.

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

Codd, E. 1983. A relational model of data for large shared data banks. *CACM 26*, 1 (Jan.). In *Milestones of Research—Selected papers 1958-1982* (CACM 25th Anniversary Issue).

Conklin, J. 1987. Hypertext: An introduction and survey. *IEEE Computer 20*, 9 (Sept.), 17–41.

Consens, M. and Mendelzon, A. 1993. Hy$^+$: A hygraph-based query and visualization system. In *Proc. ACM SIGMOD'93* (1993), pp. 511–516. Video presentation summary.

Consens, M. and Milo, T. 1994. Optimizing queries on files. In *Proc. ACM SIGMOD'94* (1994), pp. 301–312.

Consens, M. and Milo, T. 1995. Algebras for querying text regions. In *Proc. PODS'95* (1995).

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

Date, C. 1995. *An Introduction to Database Systems* (6th ed.). Addison-Wesley.

Desai, B., Goyal, P., and Sadri, S. 1986. A data model for use with formatted and textual data. *Journal of ASIS 37*, 3, 158–165.

Digital Equipment Corporation. 1991. *CDA - DDIF Technical Specification.* Digital Equipment Corporation.

Elmasri, R. and Navathe, S. 1994. *Fundamentals of Database Systems* (2nd ed.). Benjamin Cummings.

Fawcett, H. 1989. *PAT 3.3 User's Guide.* UW Centre for the New OED and Text Research, Univ. of Waterloo.

Frakes, W. and Baeza-Yates, R. Eds. 1992. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall.

Garey, M. and Johnson, D. 1979. *Computers and Intractability*. W. Freeman and Company.

Gonnet, G. and Tompa, F. 1987. Mind Your Grammar: a new approach to modelling text. In *Proc. VLDB'87* (1987), pp. 339–346.

Hull, R. and King, R. 1987. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys 19*, 3, 201–260.

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

International Standards Organization. 1991. *Information Processing - Text Composition - Standard Page Description Language (SPDL)*. International Standards Organization. ISO/IEC DIS 10180.

International Standards Organization. 1992. *Information Technology - Hypermedia/Time-based Structuring Language (HyTime)*. International Standards Organization. ISO/IEC 10744.

International Standards Organization. 1994. *Information Technology - Text and Office Systems - Document Style Semantics and Specification Language (DSSSL)*. International Standards Organization. ISO/IEC DIS 10179.2.

Kernighan, B. and Ritchie, D. 1978. *The C Programming Language*. Prentice-Hall.

Kilpeläinen, P. and Mannila, H. 1992. Grammatical tree matching. In *Proc. CPM'92* (1992), pp. 162–174.

Kilpeläinen, P. and Mannila, H. 1993. Retrieval from hierarchical texts by partial patterns. In *Proc. ACM SIGIR'93* (1993), pp. 214–222.

Kim, W. and Lochovski, F. Eds. 1989. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley.

Kuikka, E. and Salminen, A. 1995. Two-dimensional filters for structured text. *Information Processing and Management*.

Lamport, L. 1986. *LaTeX: A Document Preparation System*. Addison-Wesley.

Mackie, E. and Zobel, J. 1992. Retrieval of tree-structured data from disc. In *Proc. of the 3rd Australasian Database Conference* (1992).

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

Manber, U. and Myers, G. 1990. Suffix arrays: A new method for on-line string searches. In *Proc. ACM-SIAM'90* (1990), pp. 319–327.

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://-sunsite.dcc.uchile.cl/pub/users/gnavarro/thesis95.ps.gz`.

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

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

Sacks-Davis, R., Arnold-Moore, T., and Zobel, J. 1994. Database systems for structured documents. In *Proc. ADTI'94* (1994), pp. 272–283.

Sacks-Davis, R., Zobel, J., and Ramamohanarao, K. 1992. Advanced database systems for text retrieval. In *Proc. 3rd Australian Database Conference* (1992), pp. 1–8.

Salminen, A. and Tompa, F. 1992. PAT expressions: an algebra for text search. In *COMPLEX'92* (1992), pp. 309–332.

Salton, G. and McGill, M. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill.

Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., and Guttman, A. 1983. Document processing in a relational database system. *ACM TOIS 1*, 2 (April), 143–158.

Subrahmanian, M. and Jajodia, S. Eds. 1996. *Multimedia Database Systems*. Springer-Verlag.

Tague, J., Salminen, A., and McClellan, C. 1991. Complete formal model for information retrieval systems. In *Proc. ACM SIGIR'91* (1991), pp. 14–20.

Wu, J., Ang, Y., Lam, P., Loh, H., and Desai, A. 1994. Inference and retrieval of facial images. *ACM Multimedia Systems 2*.

APPENDIX

A. FORMAL SYNTAX AND SEMANTICS

We first define our formal model, then the syntax of the expressions we have studied ($Expr$) by an annotated abstract syntax, and finally the semantics of those operations by a function $\mathcal{I} : Expr \rightarrow \wp(\mathcal{N})$ (i.e. from expressions to sets of nodes).

A.1 Formal Model

A text database is a tuple $(\mathcal{T}, \mathcal{V}, C, N, R, Type, Segm)$, where

—$\mathcal{T} : [1..T] \rightarrow \Sigma$ is the text array. T is the size of the database (number of symbols) and Σ is the alphabet of the text.

—$\mathcal{V}$ is the finite set of hierarchies over the text, with a distinguished element $V_t \in \mathcal{V}$ (the text hierarchy).

—$C : \mathcal{V} \rightarrow \wp(\mathcal{C})$ is the set of node types of each hierarchy, we also write $C(V)$ as C_V. $\mathcal{C}$ is the finite set of node types, with a distinguished element $C_t \in \mathcal{C}$ (the text node type). It holds $\forall V_1 \neq V_2 \in \mathcal{V}, C_{V_1} \cap C_{V_2} = \emptyset$. Also, $C_{V_t} = \{C_t\}$.

—$N : \mathcal{V} \rightarrow \wp(\mathcal{N})$ is the set of nodes of each hierarchy, we also write $N(V)$ as N_V. $\mathcal{N}$ is the finite set of nodes, including special text nodes $t_{a,b}$ for each $1 \leq a \leq b \leq T$ (the text nodes). It holds $\forall V_1 \neq V_2 \in \mathcal{V}, N_{V_1} \cap N_{V_2} = \emptyset$. Also, $N_{V_t} = \{t_{a,b} / 1 \leq a \leq b \leq T\}$.

—$R : \mathcal{V} \rightarrow \wp(\mathcal{N} \times \mathcal{N})$ is the binary relationship which defines the tree of each hierarchy, we also write $R(V)$ as R_V. It holds $\forall V \in \mathcal{V}, R_V \subseteq (N_V \times N_V)$. Also, $R(V_t) = \emptyset$.

—$Type : \mathcal{N} \rightarrow \mathcal{C}$ is the type of each node. It holds $\forall V \in \mathcal{V}, \forall x \in N_V, Type(x) \in C_V$. This implies that $\forall a, b / 1 \leq a \leq b \leq T, Type(t_{a,b}) = C_t$.

—$Segm : \mathcal{N} \rightarrow [1..T] \times [1..T]$ is the segment of each node. It holds $\forall x \in \mathcal{N}, Segm(x) = (a,b) \Rightarrow a \leq b$. We also define $From$ and To to satisfy $Segm(x) = (From(x), To(x))$. Finally, we define $Segm(t_{a,b}) = (a,b)$, as expected.

We define a binary relationship $\longrightarrow$ as the union of R_V, for all $V \in \mathcal{V}$, that is $\longrightarrow = \bigcup_{V \in \mathcal{V}} R_V$. We impose the following conditions on $\longrightarrow$:

—$\forall x, y \in \mathcal{N}, x \longrightarrow^+ y \Rightarrow \neg y \longrightarrow x$, that is, loops are not allowed. Here, $\longrightarrow^+$ is the transitive closure of $\longrightarrow$.

—$\forall V \in \mathcal{V} - \{V_t\}, \exists!\ r_V \in N_V /\ \nexists x \in N_V / x \longrightarrow r_V$, that is, each hierarchy except the text hierarchy has a single root.

—$\forall x, y \in \mathcal{N}, x \longrightarrow y \Rightarrow \nexists z \neq x / z \longrightarrow y$, that is, any node has at most one parent.

—$\forall x, y \in \mathcal{N}, x \longrightarrow y \Rightarrow Segm(y) \subseteq Segm(x)$. When we operate segments as sets we interpret $Segm(x) = \{n \in Nat / From(x) \leq n \leq To(x)\}$. That is, the segment of a node includes the segment of its descendants. Nat is the set of natural numbers.

—$\forall V \in \mathcal{V} - \{V_t\}, \forall x, y \in N_V, Segm(x) \subset Segm(y) \Rightarrow x \longrightarrow^{+} y$, that is, except in the text hierarchy, if two segments of the same tree are included one into the other, then the including one is ancestor of the included.

—$\forall V \in \mathcal{V} - \{V_t\}, \forall x, y \in N_V, Segm(x) = Segm(y) \Rightarrow x \longrightarrow^{*} y \ \vee \ y \longrightarrow^{*} x$, that is, except in the text hierarchy, if two segments of the same tree are equal, then they are in a single path of the tree. Here, $\longrightarrow^{*}$ is the Kleene (transitive and reflexive) closure of $\longrightarrow$.

—$\forall V \in \mathcal{V} - \{V_t\}, \forall x, y \in N_V, \ Segm(x) \subseteq Segm(y) \vee Segm(y) \subseteq Segm(x) \vee Segm(x) \cap Segm(y) = \emptyset$, that is, there is a strict hierarchy of segments (except in the text hierarchy).

Finally, we define a binary relation in $\mathcal{N} \times \mathcal{N}$, called $\sqsubset$, to mean that the first node includes the other (do not confuse with segment inclusion). If both nodes are from the same hierarchy then the second must descend from the first one; otherwise we test for segment inclusion. Thus, $x \sqsubset y \Leftrightarrow (\exists V \in \mathcal{V} - \{V_t\} / \{x, y\} \subseteq N_V) \ ? \ y \longrightarrow^{+} x \ : \ Segm(x) \subseteq Segm(y)$. Observe that $x \sqsubset y \Rightarrow Segm(x) \subseteq Segm(y)$, but the reciprocal is not true.

A.2 Abstract Syntax and Formal Semantics

Table 4 shows the abstract syntax of the language. In this definition, we use Nat as the set of natural numbers, Int as the integers, Mat as the set of pattern-matching expressions, Pos as the language for denoting positions, and $E, E_1, E_2, E_3 \in Expr$.

Some compositions are not allowed when the operands are from different hierarchies. We indicate at the right side of each alternative, between brackets, the conditions on the intervening hierarchies for that production to be valid. The hierarchy to which the result belongs is expressed as a function $\tau : Expr \rightarrow \mathcal{V}$. Between the brackets we also indicate as simply τ which is the type to which the result of the production belongs.

We are now in position to define the semantics of the defined operations. We do so by defining a function $\mathcal{I} : Expr \rightarrow \wp(\mathcal{N})$, which interprets each expression in terms of a set of nodes.

The function $\mathcal{I}$ is defined inductively as:

—$\mathcal{I}(\mathbf{Hierarchy}(V)) = N_V$.

—$\mathcal{I}(\mathbf{Struct}(c)) = \{x \in \mathcal{N} / Type(x) = c\}$.

—Let m be a pattern-matching expression, whose result is a set of segments $(a_1, b_1)..(a_k, b_k)$. Then, $\mathcal{I}(m) = \{t_{a_i, b_i} / i \in [1..k]\}$.

—$\mathcal{I}(P \ \mathbf{collapse} \ Q) = \{t_{a_1, b_n} / \exists t_{a_1, b_1} ... t_{a_n, b_n} \in \mathcal{I}(P) \ \cup \ \mathcal{I}(Q) / (\forall i, b_i \leq a_{i+1}) \wedge \nexists t_{x,y} \in \mathcal{I}(P) \ \cup \ \mathcal{I}(Q) - \{t_{a_1, b_1} ... t_{a_n, b_n}\} / (x, y) \cap (a_1, b_n) \neq \emptyset\}$.

—$\mathcal{I}(P \ + \ Q) = \mathcal{I}(P) \cup \mathcal{I}(Q)$.

—$\mathcal{I}(P \ - \ Q) = \mathcal{I}(P) - \mathcal{I}(Q)$.

—$\mathcal{I}(P \ \mathbf{is} \ Q) = \mathcal{I}(P) \cap \mathcal{I}(Q)$.

$Expr$	$\longrightarrow$	$\textbf{Hierarchy}(V)\ [V \in \mathcal{V} - \{V_t\}, \tau = V]$
	$\mid$	$\textbf{Struct}(c)\ [c \in \mathcal{C} - \{C_t\}, \tau = V/c \in C_V]$
	$\mid$	$\textbf{Match}(m)\ [m \in Mat, \tau = V_t]$
	$\mid$	$(E_1\ \textbf{collapse}\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) = V_t]$
	$\mid$	... (other operations to manipulate matches)
	$\mid$	$(E_1\ +\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t]$
	$\mid$	$(E_1\ -\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t]$
	$\mid$	$(E_1\ \textbf{is}\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t\ ]$
	$\mid$	$(E_1\ \textbf{same}\ E_2)\ [\tau = \tau(E_1)]$
	$\mid$	$(E_1\ \textbf{with}(k)\ E_2)\ [k \in Nat, \tau = \tau(E_1)]$
	$\mid$	$(E_1\ \textbf{withbegin/withend}(k)\ E_2)\ [k \in Nat, \tau = \tau(E_1) \neq \tau(E_2)]$
	$\mid$	$(E_1\ \textbf{in}\ E_2)\ [\tau = \tau(E_1)]$
	$\mid$	$(E_1\ \textbf{beginin/endin}\ E_2)\ [\tau = \tau(E_1) \neq \tau(E_2)]$
	$\mid$	$([s]\ E_1\ \textbf{in}\ E_2)\ [s \in Pos, \tau = \tau(E_1)]$
	$\mid$	$([s]\ E_1\ \textbf{beginin/endin}\ E_2)\ [s \in Pos, \tau = \tau(E_1) \neq \tau(E_2)]$
	$\mid$	$(E_1\ \textbf{parent}(k)\ E_2)\ [k \in Nat, \tau = \tau(E_1) = \tau(E_2) \neq V_t]$
	$\mid$	$([s]\ E_1\ \textbf{child}\ E_2)\ [s \in Pos, \tau = \tau(E_1) = \tau(E_2) \neq V_t]$
	$\mid$	$(E_1\ \textbf{after/before}\ E_2\ (E_3))\ [\tau = \tau(E_1)]$
	$\mid$	$(E_1\ \textbf{after/before}(k)\ E_2\ (E_3))\ [k \in Nat, \tau = \tau(E_1)]$

Table 4. Abstract syntax.

—$\mathcal{I}(P\ \textbf{same}\ Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / Segm(x) = Segm(y)\}$.

—$\mathcal{I}(P\ \textbf{with}(k)\ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / y \subset x\}| \geq k\}$.

—$\mathcal{I}(P\ \textbf{withbegin}(k)\ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / From(y) \in Segm(x)\}| \geq k\}$.

—$\mathcal{I}(P\ \textbf{withend}(k)\ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / To(y) \in Segm(x)\}| \geq k\}$.

—$\mathcal{I}(P\ \textbf{in}\ Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / x \subset y\}$.

—$\mathcal{I}(P\ \textbf{beginin}\ Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / From(x) \in Segm(y)\}$.

—$\mathcal{I}(P\ \textbf{endin}\ Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / To(x) \in Segm(y)\}$.

—$\mathcal{I}([s]\ P\ \textbf{in}\ Q) = \bigcup_{y \in \mathcal{I}(Q)} \{x \in \mathcal{Z}_y / \mathcal{S}(s, x, \mathcal{Z}_y)\}$. Here, $\mathcal{S} : S \times \mathcal{N} \times \wp(\mathcal{N}) \rightarrow \{true, false\}$ is the interpretation of the position language S, which says whether the left-to-right position of a node in a set of nodes is acceptable by the specification of s. This position is only well defined when none of the segments includes another, which is the case in $\mathcal{Z}_y$, that we define as $\mathcal{Z}_y = \{x \in \mathcal{I}(P) / x \subset y\ \wedge\ x \in maxim(z \in \mathcal{I}(P) / z \subset y\ \vee\ y \not\subset z)\}$. $maxim$ selects the maximal nodes of a set, i.e. $maxim(X) = \{x \in X /\ \nexists y \in X / x \subset y\}$.

—$\mathcal{I}([s]\ P\ \textbf{beginin}\ Q) = \bigcup_{y \in \mathcal{I}(Q)} \{x \in \mathcal{Z}_y / \mathcal{S}(s, x, \mathcal{Z}_y)\}$. Here, $\mathcal{Z}_y = \{x \in \mathcal{I}(P) / From(x) \in Segm(y)\ \wedge\ x \in maxim(z \in \mathcal{I}(P) / Segm(z) \not\supset Segm(y))\}$.

—$\mathcal{I}([s]\ P\ \textbf{endin}\ Q) = \bigcup_{y \in \mathcal{I}(Q)} \{x \in \mathcal{Z}_y / \mathcal{S}(s, x, \mathcal{Z}_y)\}$. Here, $\mathcal{Z}_y = \{x \in \mathcal{I}(P) / To(x) \in Segm(y)\ \wedge\ x \in maxim(z \in \mathcal{I}(P) / Segm(z) \not\supset Segm(y))\}$.

—$\mathcal{I}(P\ \textbf{parent}(k)\ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / x \longrightarrow y\}| \geq k\}$.

—$\mathcal{I}([s]\ P\ \textbf{child}\ Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / y \longrightarrow x \wedge \mathcal{S}(s, x, \{z \in \mathcal{N} / y \longrightarrow z\})\}$.

—$\mathcal{I}(P\ \textbf{after}(k)\ Q\ (C)) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / 0 < From(x) - To(y) \leq k\ \wedge\ minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\}$.

—$\mathcal{I}(P\ \textbf{after}\ Q\ (C)) = \bigcup_{y \in \mathcal{I}(Q)} first(\{x \in \mathcal{I}(P) / From(x) > To(y)\ \wedge\ minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\})$. Here, $first : \wp(\mathcal{N}) \rightarrow \mathcal{N}$ selects the node in the set with lowest value of $From$, and if there are more than one, the maximal. If all the nodes are from the same hierarchy, this criterion gives exactly one node.

—$\mathcal{I}(P\ \mathbf{before}(k)\ Q\ (C)) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / 0 < From(y) - To(x) \leq k\ \wedge\ minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\}$.

—$\mathcal{I}(P\ \mathbf{before}\ Q\ (C)) = \bigcup_{y \in \mathcal{I}(Q)} last(\{x \in \mathcal{I}(P) / From(y) > To(x) \wedge minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\})$. *last* is analogous to *first*, selecting the highest value of To, or the maximal if they are the same.