A Language for Queries on Structure and Contents of Textual Databases*

Gonzalo Navarro Ricardo Baeza-Yates

Dept. of Computer Science
University of Chile
Blanco Encalada 2120,
Santiago, Chile
{gnavarro,rbaeza}@dcc.uchile.cl

Abstract

We present a model for querying textual databases by both the structure and contents of the text. Our goal is to obtain a query language which is expressive enough in practice while being efficiently implementable, features not present at the same time in previous work. We evaluate our model regarding expressivity and efficiency. The key idea of the model is that a set-oriented query language based on operations on nearby structure elements of one or more hierarchies is quite expressive and efficiently implementable, being a good tradeoff between both goals.

1 Introduction

Textual databases are deserving more and more attention, due to their multiple applications: libraries, office automation, software engineering, automated dictionaries and encyclopedias, and in general any problem based on keeping and retrieving textual information [11].

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

Any information model for a text database should comprise three parts: text, structure, and query language. It must specify how is the text seen (i.e. character set, synonyms, stopwords, hidden portions, etc.), the structuring mechanism (i.e. markup, index structure, type of structuring, etc.), and the query language (i.e. what things can be asked, what the answers are, etc.).

The purpose of any system related to information retrieval is to help the users of a database to find what they need. Textual database are not as relational databases [8], 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 is satisfied with the answer. Anything we can do to help users to find what they want is worth considering.

Traditionally, textual databases have allowed their users to search their contents (words, phrases, etc.) or their structure (e.g. by looking at a table of contents). These are two mechanisms by which users can find what they need.

Another interesting fact is that human beings have "visual memory", e.g. they may remember that what they want was typed in *italics*, short before a figure that said something about "earth". Searching for the word "earth" may not be a good idea, as well as searching all figures or all the text in italics. What really would help to exploit visual memory would be a language in which we can say "I want a text on italics, near a figure containing the word 'earth' ". This query mixes content and structure of the database.

Mixing contents and structure in queries allows us to pose very powerful queries, being much more expressive than each mechanism by itself. By using a query language that integrates both types of queries, we can potentiate the retrieval quality of textual databases.

The aim of this paper is to present a model to structure and query textual databases, which is expressive enough and efficiently implementable. There is not at this time, to the best of our knowledge, any approach satisfying both goals. A more complete presentation of this work can be found in [19].

The query language we present is not necessarily intended for final users, rather it is an operational algebra onto which one can map a more user-oriented query language.

This paper is organized as follows. In section 2, related work is reviewed. In section 3, our model is presented, in terms of a data model and the operations allowed for queries. In section 4, we evaluate the model regarding expressivity and efficiency. Finally, in section 5, our conclusions and future work directions are outlined.

2 Related Work

In this section we briefly review previous approaches to the problem of querying a textual database. We first mention the traditional ones, and then cover novel ideas.

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 [8] to include text management [24, 9]; the many traditional models of information re-

*This work has been supported in part by grants FONDECYT (Chile) 1940271 and 1950622.

trieval (e.g. the boolean model, the probabilistic model, the bit-vector model, the full-text model, etc.) [23, 11]; hypertext [5] and semantic networks [13, 25]; and object-oriented databases adapted to manage text [16, 3].

None of these approaches satisfy our goals of mixing structure and contents in queries. Some of them do not allow to express rich enough structures, others are too oriented to contents, and others to structure. Finally, although object-oriented database can be extended to successfully combine both areas, they are too general and do not exploit the semantics involved in structuring. They represent the structure merely as a network and have a query language oriented to those graphs, making very inefficient queries that would be simple if we knew the inclusion semantics involved in the structure (e.g. costly path-expressions on part-of hierarchies can be expressed as simple segment inclusion in many cases). See [6] for an excellent work on this topic.

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

In [21] it is 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. Hence, each subsystem focuses on the part of the query in which specializes (e.g. [6] integrates an object-oriented database with a structured text engine).

We rely on this approach. We design a language which is focused on exploiting structure- and text-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.

On the other hand, we do not address the 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). See [21] for some ideas on this problem.

2.2 Novel Approaches

These approaches are characterized by generally imposing a hierarchical structure on the database, and by mixing queries on contents and structure. Although this structuring is simpler than, for example, hypertext, even in this simpler case the problem of mixing contents and structure is not satisfactorily solved.

We present a sample of novel models, which cover many different approaches to solve this problem under the stated conditions. See [17] for another survey.

The Hybrid Model [1]: modelizes a textual database as a set of documents, which may have fields. 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 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 flat and not fully compositional. This model can be implemented very efficiently.

PAT Expressions [22, 10]: sees only match points, which are used to define *regions*. Regions are defined by match expressions that specify how their endpoints are. Each region represents a set of disjoint segments. This allows dynamic definition of regions, and to match all queries on regions to queries on matches. The need to avoid overlapping regions cause a lot of troubles and lack of orthogonality in the language. This language achieves high efficiency at the cost of some restrictions, which for some applications are reasonable.

Overlapped Lists [4]: 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 the same way. The implementation of this model can be as efficient as that of PAT expressions.

Lists of References [18]: is a general model to structure and query textual databases, including also hypertext-like linkages, attribute management and external procedures. The structure of documents can be hierarchical (no overlaps), but answers to queries are flat (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 to integrate in an elegant way answers to queries to 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 [12]: is in fact a data manipulation language. To express database schemas a context-free grammar is used, 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 powerful operations to transform trees. This approach is extremely powerful, and it is shown to be relationally complete. However, it is hard to implement efficiently [2].

Tree Matching [15]: 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 match, not only the root. The complexity of the algorithms is studied, showing that the only case in which the problem is of polynomial time is when no logical variables are used and if the matches have to satisfy the left-to-right ordering in the pattern. Even in the polynomial case, the operations have to traverse the whole database structure to find the matches.

3 A New Model for Querying Structured Text

In this section we give an informal description of our model. A formal presentation can be found in [19].

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 expressivity 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 what we want across the whole tree of the structure (e.g. [14]),.

Since we want to define a fully compositional query language, we can consider query expressions as syntax trees, where the nodes represent operations to perform and the subtrees their operands.

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 element.

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

Our solution is an algebra over sets of nodes (each node is a structural element of the database, e.g. a particular chapter or figure). That means that the operations take sets of nodes and return a set of nodes. These sets of nodes are subsets of the set of all nodes of the tree of the database. The only place in which we pose a text matching query or name a structural component should be at the leaves of the syntax tree of queries. These leaves must be solved with some sort of index, and converted to a set of nodes. Thereafter, all operators deal with these sets of nodes and produce new sets. Figure 1 shows the main concepts, which are refined later, to detail the query language and to draw a general software architecture comprising this model.

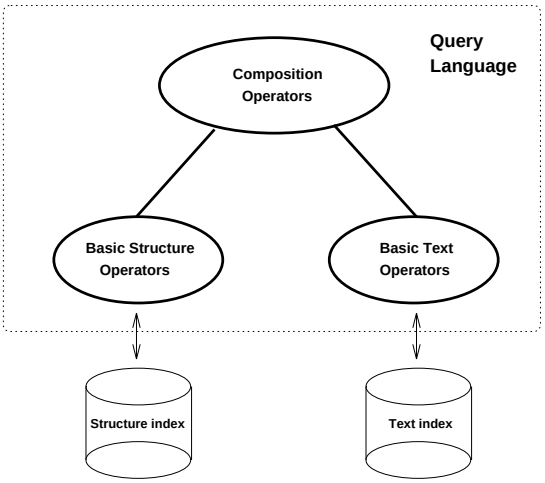

Figure 1: Initial diagram of how our model operates.

With this approach, we use indices to retrieve the nodes that satisfy a text matching query, or the nodes corresponding to a given structural component, also called "constructor" (e.g. chapters). These nodes must be obtained without traversing the whole database.

Once we have converted the leaves of the query syntax tree into sets, all the other operations take the 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 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 nodes and segments. The segments are pairs of numbers representing contiguous portions of the text (e.g. the segment of a chapter includes all its text). This coupling allows us to use efficient data structures to arrange the nodes by looking at their segments (for example, forming a tree). In other approaches [15, 12], 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 operations should be defined in such a way that they only need to match nodes from both operands that are more or less proximal. When this happens, we can easily apply divide-and-conquer techniques to drastically limit the area in which we must search for matching nodes.

If we can efficiently convert text matching and named structural components into well-arranged sets of nodes, and all operators can efficiently work with the arranged sets and produce arranged sets, then we will have an efficient implementation.

This schema allows us to have more than one structure hierarchy, if they are independent.

On the other hand, we must show that many interesting operators are in fact of the kind we need, i.e. they operate on nearby nodes and all what they need to operate 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 that kind of operations.

3.2 Data Model

A text database is composed of two parts:

- Text, which is seen as a (long) sequence of symbols. 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. Additionally, symbols may be characters, words, etc.

- Structure, which is organized as a set of independent (orthogonal) hierarchies. 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.

Removing the markup from the document 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 display 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 a query 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 contents 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 to build new hierarchies, delete and modify 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.

Each hierarchy (or tree) is called a *view*, which as its name suggests, is an independent way to see the text (e.g. chapters / sections / paragraphs and pages / lines). The root of each view is a special node considered to comprise the whole database.

Each view has a set of *constructors*, which denote types of nodes of the corresponding tree. Examples of constructors are page, chapter and section. The sets of constructors of different views are disjoint.

Each node of the tree corresponding to a view has an associated constructor, and 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).

Any set of disjoint segments can be seen as belonging to a special *text view*, where the nodes belong to a *text constructor* and have flat structure (all nodes at the second level of the tree). Thus, the *text view* has one node for each possible segment of the text. The idea is to use that view to model pattern-matching queries, which we impose to have flat structure. This restriction is not essential, since those pattern-matching expressions could perfectly well generate a nested structure. However, we assume that the structure is flat for some operations on pattern-matching queries, which would not be applicable if the structure was not flat.

3.3 Query Language

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

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 implementable).

Each set produced by evaluating a query is a subset of some view. Each element of this set is a single node, representing a single segment.

We decided not to merge nodes from different views in a single result for two reasons: first, it is not clear, views 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 the approach of [4], 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 on the basis of their *context* in the view tree, or the trees of the operands, much like in [15].

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 now the operations we consider are enough for a large set of applications, and suitable to be efficiently implemented. As we said before, this set is not exclusive nor essential.

In Figure 2 we outline the schema of the operations. There are basic extraction operations (forming the basis of querying on structure and on contents), and operations to combine results from others, which are classified in a number of groups: those which operate by considering included elements, including elements, nearby elements, to manipulate sets and by direct structural relationships.

Matching sublanguage: Is the only one which accesses the text contents of the database, and is orthogonal to the rest of the language.

- Matches: The matching language generates a set of non-overlapping segments, which are introduced in the model as belonging to the text view, as explained before. For example, `"computer"` generates the flat tree of all segments where that word appears in the text. Note that the matching language could allow much more complex expressions.
- Operations on matches: Are applicable only to subsets of the text view, 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 [19] for some alternatives. As an example, we propose M **collapse** M', which superimposes both sets of matches, merging them when an overlap results, or M **subtract** M', which deletes from M the points from segments present in M'.

Basic structure operators: Are the other kind of leaves of the syntax tree, which refer to basic structural components.

- Name of constructor: ("**Constr**" queries). Is the set of all nodes of the given constructor. For example, `chapter` retrieves all chapters in a book.
- Name of view: ("**View**" queries). Is the set of all nodes of the given view. For example, `Formatting` retrieves the whole view related to formatting aspects. The same effect can be obtained by summing up ("+" operator) all the constructors of the view.

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.

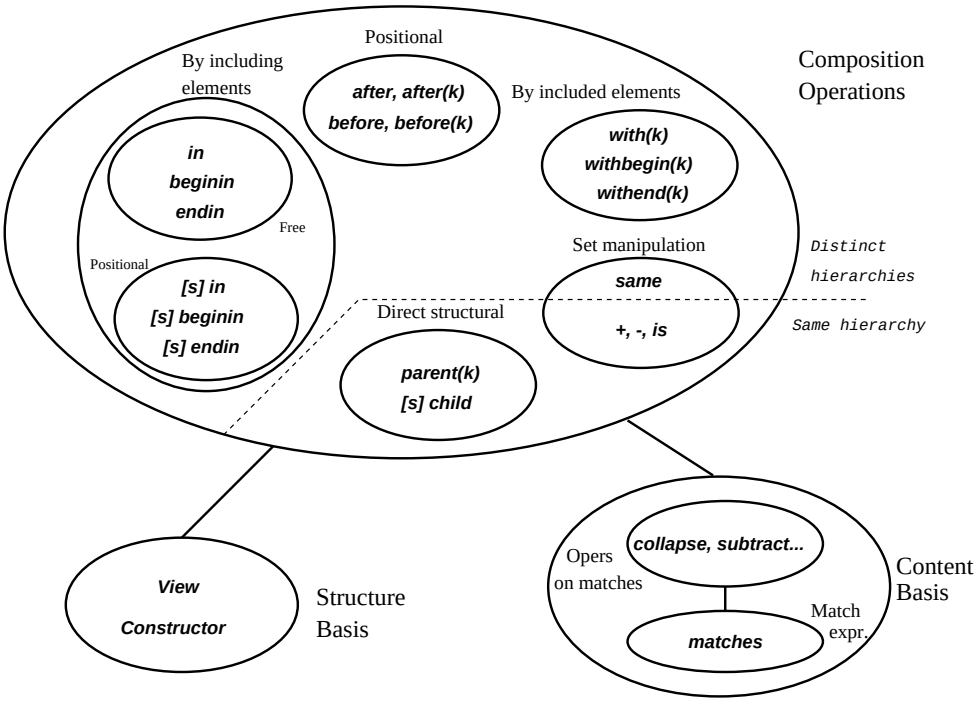

Figure 2: The operations of our model, classified by type.

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, `chapter` **beginin** `italics` are the chapters that begin when the *italic* font is active.

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 expressing 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 include other paragraphs, only the top-level ones are considered, and those partially included in a page are discarded.

[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 times or more.

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 pages or more (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 relationships of direct parentship in the tree of the view. Both operands must be from the same view, which cannot be the text view.

[s] P child Q: Is the set of nodes of P which are children (in the view tree) 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). Note that **child** is not essential, since $[s]\ P$ **child** $Q = P$ **is** $([s]\ View$ **in** $Q)$, but this alternative is much more expensive.

P parent(k) Q: Is the set of nodes of P which are parents (in the view tree) 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.

Positional 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 higher one). In order for a node of P to be considered a candidate for a Q-node, the minimal node of C that contains it must be the same than that of the Q-node, or must not exist in both cases. For example, `table` **after** `figure (chapter)` retrieves tables which are nearest to a figure preceding them, 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 characters (or words, depending on the view that we have on the text), 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 view (which must not be the text view).

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, use P **subtract** (P **same** Q).

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 segment is the same segment of a node in Q. P and Q can be from different views. 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 views, since otherwise they are the same as their full segment counterparts.

Except for **child** and **View**, the operators are not redundant. One can consider that there are too many operands, but recall that we do not propose a specific query language, rather we point out a number of operators that are efficiently implementable within our approach.

3.3.2 Examples

Now we present some examples of the use of these operators, to give an idea of what kind of queries can we pose with this language.

Suppose we have a view V with constructors `book`, `introduction`, `bibliography`, `chapter`, `appendix`, `section`, `paragraph` and `formula`. A `book` has an `introduction`, a number of `chapters`, a `bibliography` and an `appendix` that has `sections`. `chapters` also have `sections` and `sections` have more `sections` inside them, and `paragraphs`. We also have `figure` and `table`, which can be children of a `section` or a `chapter`. A table is divided in `rows`, and these in `columns`. 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 view V' with `volume`, `page` and `line`. We have still another view VP for presentation aspects, e.g. `underline`, `emphasize`, `font`, etc.

Suppose also that we have a simple matching language, in which it is only possible to find a given word.

- `italics` **before**`(100) (figure` **with** `"graphs") (page)` is the query we wanted in the Introduction.
- `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** if you want 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.

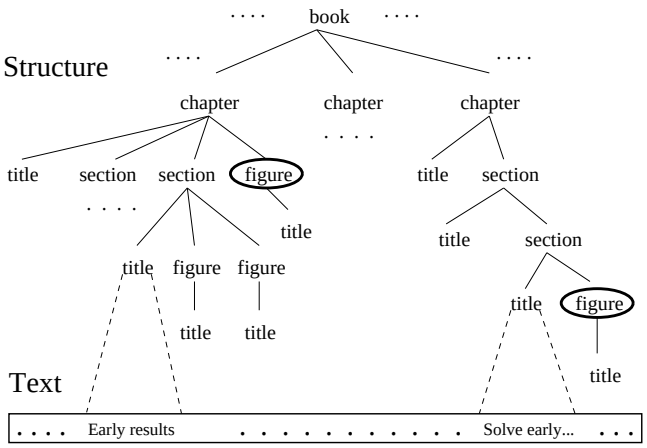

Figure 3: Illustration of the effect of the query `[last] figure in (chapter with (section with (title with "early")))`. The circles indicate selected nodes.

3.4 A Software Architecture

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

Users should interact with our system via an interface, in which they define what they want in a friendly language (see [15] for an example of a friendly language oriented to querying structured databases). That interface should then 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. linearizes the tree into a sequence of operations to perform). The leaves of the query tree involve extracting components of the hierarchy by name (constructors), and text matching subexpressions. The first ones are solved by accessing the index on structure to extract the whole set of nodes from that constructor (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 to the interface.

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

The text engine is in charge of offering a text pattern-matching language, in which it accepts queries and returns the corresponding list of segments; to keep the indices it needs for searching; and to present a filtered version of the text file to upper layers, in order to retrieve the contents of a submitted text segment.

If the text engine is a completely separate subsystem, two separate indexing processes can 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 Figure 4 for a diagram of how a complete system based on this schema should be. The "document layer" is intended to support more sophisticated document management, such as collections of documents, etc.

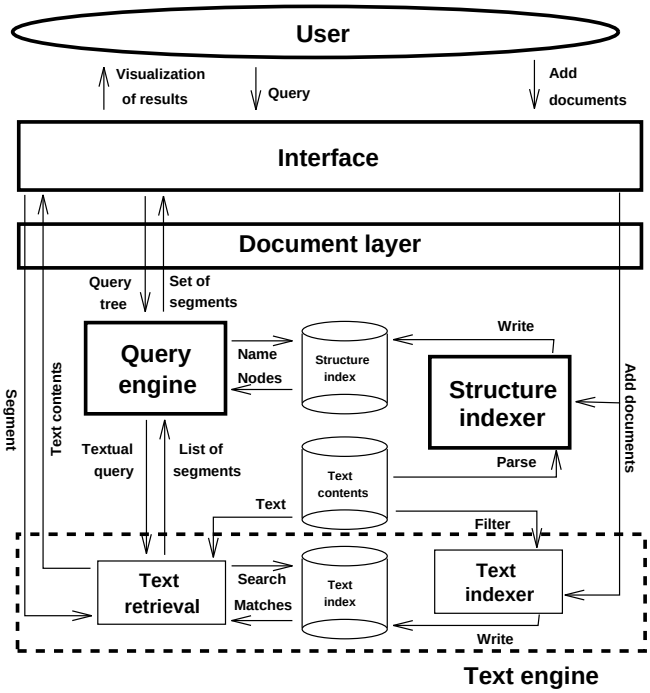

Figure 4: The architecture of a system following our model.

4 Evaluation

In this section we briefly present the results of evaluating this model, both in expressivity and efficiency. The reader is referred to [19] for details.

4.1 Expressivity

We have compared our model against the novel models we surveyed here. We defined formally the semantics of our operations and compared our model against each other, to determine which features from ours can be represented in others and vice versa. Later, we defined an informal framework to situate similar models [19, 20].

We present in Figure 5 a graphical version of this comparison. We identify the main points about expressivity, and represented each model as a set containing the points it reasonably supports. The *p-strings* model is not included, because is a data manipulation language.

From the figure, we can see that the main features lacking in our model are tuples, semijoin by contents and the possibility of having overlaps and combined nodes in the result set of a query. We believe that none of them can be included without degrading the performance. The set we

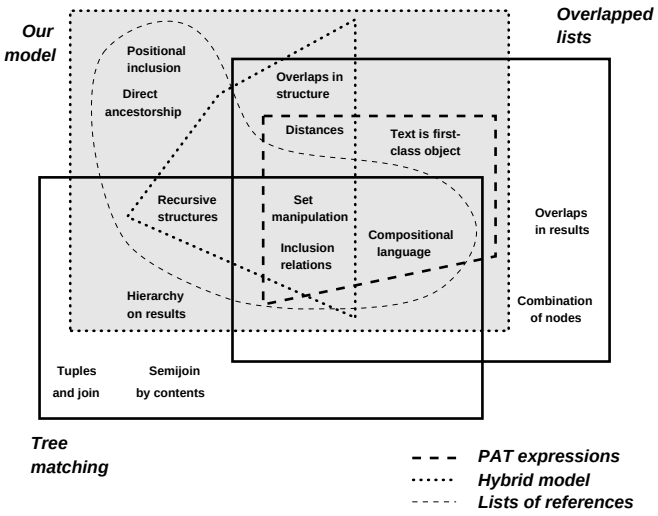

Figure 5: A graphical representation of the comparison made between models.

include is enough for a large class of applications. Some of the lacking features are better included by integrating this model with another one (e.g. an object-oriented database).

4.2 Efficiency

We have defined algorithms and data structures to implement our model, and analyzed their worst-case behavior. We also have implemented these algorithms in a prototype, which we used to obtain average times running the model with real data.

The sets of nodes to operate are arranged as trees, by looking at their embedding in the hierarchy of the view they belong to. Several versions of the algorithms have been studied, being the best a merge-like approach that traverses both trees in synchronization.

Two implementations are analyzed here: a full evaluation version computes the whole set of answers at once; while a lazy evaluation version 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.

While the lazy version forces an order of evaluation that is not always optimal and hence has higher complexity, it can compute only part of the result, so which one is better has to be experimentally determined.

The results are summarized in table 1.

The performance of **View** and **Constr** queries depend on the indexing scheme, being normally linear in the size of the result.

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

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 that machine. A rough approximation to this is $(2q-1) \times average\ operand\ size$, 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 expanded.

Operation	Full	Lazy
$+,-$	n	$n \min(d,h)$
is/same	n	n
in	$\min(n, d^2h)$	$\min(n, d^2h)$
beginin/endin	$\min(n, d^2h)$	$\min(n+dh, d^2h)$
$[s]$***in**	$n \min(d,h)$	$n \min(d,h)$
with*(k)	n	$n \min(n, k+dh)$
$[s]$**child**	n	n
parent(k)	n	$n \min(d,h)$
after/before	$n \min(n,dh)$	$n \min(n,dh)$

Table 1: 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.

These good complexity results are possible thanks to our approach of coupling nodes with segments, which allows us to readily apply divide-and-conquer techniques for obtaining the whole set of solutions to a query. The ideas of a set-oriented query language, a data structure in which we can easily separate ranges of segments, and the reduction of all queries to operations on proximal nodes lead us to an implementation where the amortized cost per retrieved element is, in many cases, constant.

5 Conclusions and Future Work

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

Then, we have defined a model for structuring and querying textual databases that is expressive enough and efficiently implementable. This language is not meant to be accessed by final users, but to constitute the operational algebra.

Finally, we have evaluated our model in terms of expressivity and efficiency. The model has been shown to be competitive in expressivity, 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 in the tests, what situates this model close in efficiency to those which have much less expressivity.

See Figure 6 for a graphical (and informal) comparison of similar models when taking into account both efficiency and expressivity. Note that we have included *p-strings* in this drawing, assuming an expressivity 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). Note that, as any quantization of concepts, this comparison is subjective. Nevertheless, it does give an idea of where our model is.

There are a number of research directions related with the model:

- Exploration of the possibilities offered by our model in order to find more interesting operators which lie into our philosophy (being thus efficiently implementable).
- Definition of a query language suitable for end users, possibly visual, to map onto our operational algebra.

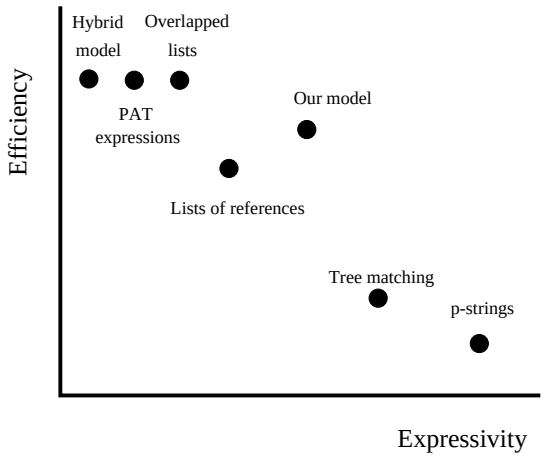

Figure 6: A comparison between similar models, regarding both efficiency and expressivity.

- Integration between this kind of model and others, such as the relational or the traditional ones of information retrieval. This issue has not been considered here, since we focus on the structure problem. See [21] for some ideas on this area.
- 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 formal framework in which to compare expressivity is needed. The long-term goal should be a formal and sound hierarchy like what can be found in the area of formal languages (see [20, 7] for some examples).

References

[1] R. Baeza-Yates. An hybrid query model for full text retrieval systems. Technical Report DCC-1994-2, Dept. of Computer Science, Univ. of Chile, 1994.

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

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

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

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

[6] M. Consens and T. Milo. Optimizing queries on files. In *Proc. ACM SIGMOD'94*, pages 301–312, 1994.

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

[8] C. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Massachusetts, 6th edition, 1995.

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

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

[11] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1992.

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

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

[14] P. Kilpeläinen and H. Mannila. Grammatical tree matching. In *Proc. CPM'92*, pages 162–174, 1992.

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

[16] W. Kim and F. Lochovski, editors. *Object-Oriented Concepts, Databases and Applications.* Addison-Wesley, Reading, Massachusetts, 1989.

[17] A. Loeffen. Text databases: A survey of text models and systems. *ACM SIGMOD Conference. ACM SIGMOD RECORD*, 23(1):97–106, Mar. 1994.

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

[19] G. Navarro. A language for queries on structure and contents of textual databases. Master's thesis, Dept. of Computer Science, Univ. of Chile, Apr. 1995.

[20] G. Navarro and R. Baeza-Yates. Expressive power of a new model for structured text databases. In *Proc. PANEL'95*, Aug. 1995. Canela, Brazil.

[21] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents. In *Proc. ADTI'94*, pages 272–283, 1994.

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

[23] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

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

[25] J. Tague, A. Salminen, and C. McClellan. Complete formal model for information retrieval systems. In *Proc. ACM SIGIR'91*, pages 14–20, 1991.