A Language for Queries on Structure and Contents of Textual Databases

by

Gonzalo Navarro

A thesis
presented to the University of Chile
in fulfilment of the
thesis requirement for the degree of
Masters in Computer Science

Advisor: **Ricardo Baeza-Yates**

Evaluation Commitee: Jorge Olivos
Patricio Poblete
Nivio Ziviani

SANTIAGO - CHILE

April , 1995

A Betina,
sin quien nada tendría sentido.

Abstract

This thesis focuses on the problem of finding a suitable query language for hierarchically structured textual databases.

The problem about current approaches is that there is no consensus on how the structuring model and the query language should be, and that they focus strongly either on expressivity or on efficiency issues, but not on both at the same time. The approaches which are strong in one point are weak in the other. Moreover, there is no formal and complete foundation to analyze the expressivity of these languages.

The goal of this thesis is to find a structural model and a query language that is expressive and efficiently implementable, achieving a good compromise between the two extremes.

In order to achieve this goal, a number of steps have been carried out. In the first place, a comprehensive study and evaluation of previous work on the field has been done. Then, a structuring model and query language with the desired characteristics has been defined. Its expressivity has been compared against similar models, formally and practically. An informal framework to compare the expressivity of similar models has been defined. Then, we focused on implementation. Algorithms have been defined and their worst-case space and time complexity analyzed for all operations, in many different versions for implementing indices and operations. Finally, a real prototype has been developed implementing the proposed model, to evaluate heuristics and draw average running times.

This work leads to the conclusion that a set-oriented query language based on operations on nearby structure elements of one or more hierarchies is quite expressive and efficiently implementable. It also gives an idea of up to where the expressivity can be enriched without degrading the performance. Finally, it suggests some research directions.

This work makes a step in the direction of obtaining a unifying perspective on how a query language for textual databases should be, what expressive power should it have and how well can it be implemented. All this is necessary to put the emerging area of textual databases in the place it deserves in Computer Science.

Acknowledgments

Not only this work, but the whole way of life that has became a reality for me in the last year, would have been impossible without the sincere and disinterested friendship and support received from a number of people. I am indebted to all of them.

First of all, my wife Betina, that left all to follow me in this adventure. Without she, my life would be by far less interesting to live.

Then I should mention Jorge Olivos, who knew me at ESLAI, and has been pushing me since then to leave my job and come to Chile, to enjoy serious research. He also made the first steps in giving me an opportunity to come, and has been always ready to bring me his friendship and help.

A special note for Ricardo Baeza, who not only trusted me from the first time, has been an excellent advisor, gave me the main financial support, and helped me in every imaginable way, but has been also a constant friend and guide in this new life.

I want also to acknowledge Patricio Poblete, for his financial support (without even knowing me), his friendship and the stimulating lectures and joint work I enjoyed with him.

I would also like to mention a number of friends who made my life more enjoyable here: Rodrigo, Nancy, Gastón, Eduardo, Marcelo, Christian... and also my old friends, that are still with me by e-mail and with who I have enjoyed a lot: Sergio, Pablo, Hernán, Andrés, Mariano... and to my family, who is always proud of me, even when I do nothing remarkable.

My gratitude to my thesis committee, who read the whole thesis and made a number of useful comments that bettered the work in many ways: Jorge Olivos (again), Patricio Poblete (again) and Nivio Ziviani.

Finally, I want to acknowledge the Postgraduate School, for allowing me to use English for this thesis, and see how this experiment works.

Last but not least, FONDECYT grants 1940271, 1930765 and 1950622, which partially supported this work, are gratefully acknowledged.

Contents

List of Figures

List of Tables

Chapter 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 [FBY92, chapter 1].

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.

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 [Dat95], 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 to search their contents (words, phrases, etc.) or their structure (e.g. by looking at a table of contents), but not both at the same time.

An 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.

This way, we have that the information model of textual databases comprises both content and structure, and that we want to be able to query both aspects of the database. However, both aspects present their own special features.

On one hand, the "contents" of the database is not formatted, but appears in natural language form. This means that no traditional methodology relying on formatted data (e.g. the relational model) is suitable for extracting the desired information; and that no methodology assuming uninterpreted data objects and relying only on their (formatted) attributes (e.g. multimedia databases [BRG88]) is powerful enough to express the rich information model represented by text. The required information has to be extracted from the text, but not in a rigid way (see also [SDAMZ94]).

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 the structure can be done. There is a tradeoff between providing a powerful structuring model and implementing it efficiently.

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 aim of this work is to present a model to structure and query textual databases, which is expressive and efficiently implementable. There is not at this time, to the best of our knowledge, any approach satisfying both goals. In order to compare expressivity, it is necessary to draw a framework in which to situate all approaches, which does not exist yet. In order to compare efficiency, it is necessary to describe the algorithms to build the indices and to process queries, to analyze their worst-case behavior in terms of space and time, and to draw experimental results from an implementation of the model.

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 thesis is organized as follows. In Chapter 2, related work is reviewed. In Chapter 3, our model is informally presented, in terms of the data model and the operations allowed for queries. In Chapter 4, we analyze the expressivity obtained by the defined operators: we define the operations formally, compare this model against each similar model we reviewed, and draw a framework on expressivity to situate similar models. In Chapter 5, we outline algorithms and data structures to efficiently implement this model, and analyze its worst-case behavior in terms of time and space. In Chapter 6, a real implementation is presented, which is used to draw real numbers on average times to solve sample queries, and to evaluate heuristics. Finally, in Chapter 7, our conclusions and future work directions are outlined.

Chapter 2

Related Work

In this chapter we cover previous approaches to the problem of querying a textual database. We first review the traditional ones, and then cover more in depth novel ideas.

2.1 Traditional Approaches

This chapter presents briefly the classical approaches. In these, no consensus on the structuring model exists, and the querying on contents is divorced from the querying on structure.

The Relational Model: The relational model [Dat95] expresses the relationships present in a database by a fixed structure of tables, in which the data is organized. By developing an efficient and versatile set of operators to manipulate those tables, this model has successfully been applied to a wide range of information management problems.

However, this model is not suitable for expressing the fuzzy, complex and highly variable structuring present in a textual database [GT87, KM93] (e.g. a concept as basic in a hierarchical structure as ancestorship involves transitive closure, which the relational model cannot express), not to mention the extraction of information from contents, which lies completely outside the scope of this information model. Some proposals for integrating the relational model with a textual query language can be found in [SSL$^+$83, DGS86].

The Traditional IR Model: The traditional IR (Information Retrieval) model [SM83, Sal86, SM89] was the first in recognizing the particular information requirements posed by textual databases, and the need to create a model oriented to text. In this model, a database is organized as a set of documents, which are assigned *keywords*, that is, words or phrases meant to describe the semantic contents of the document. Queries are in terms of those keywords, and by examining the correlation between the words of the query and the keywords of the document, the relevance of the document for that query is established. Therefore, the answer to the query consists of a sequence of documents (ranked according to the computed relevance). There are many variations on this topic, for example relevance ranking, the boolean model, the probabilistic model, the vector model, etc. [FBY92, chapters 11, 12, 14 and 15].

Along with queries about keywords, we can also query on contents. The only structure allowed on contents consists of non-nested, non-overlapped "fields", regions which cover the whole document. Those fields can only be used to restrict the areas in which match points are to be found. The problem with this approach is that the fine structure of documents is lost, since they are seen as "black boxes" whose only description are their keywords, their (restricted) fields and their content. This approach is acceptable in multimedia databases [BRG88], in which objects as audio segments, video clips or images are assigned *descriptors*, to which the query can refer; but the text has a much richer structure, which with this approach is lost.

The Full Text Model: Another approach consists of querying only by contents, in which a query is a *pattern*, which is searched in the whole database, answering with the documents and the text positions in which the pattern matches the text [FBY92, chapter 10] and [BY94]. This search may not use any additional index, in which case the search has to traverse the whole text database; or it may use some kind of index (e.g. inverted files [FBY92, chapter 3], signature files [FBY92, chapter 4], etc.). If the database is large, the last option is the only acceptable. A novel index is the "suffix array" or "PAT array", which allows, among other things, to find any prefix in a time which is logarithmic on the size of the database [FBY92, chapter 5] and [MM90]. The problem with this search by contents is that it is not possible to query on the structure of documents. Most commercially available products combine full text retrieval with the IR model.

Hypertext: In hypertext [Con87], the database is organized as a graph where nodes are small portions of the database, and edges connect nodes which are related by some design criterion. In this case, the idea is not to use querying but a navigational mechanism across the database, with some association semantics. Edges may not only express associations by semantic similarity, but also the structure of the text, cross references, etc. Hypertexts model query by structure well, but not by contents; moreover, not always a navigational approach is acceptable. Recently, some models combining a semantic network [HK87] with structured text have appeared [TSM91], resulting in a hypertext with some facilities to query on the text and its structure.

Related to this we can mention also graph query languages [CM93] and object-oriented databases [KL89]. Some attempts have been made to integrate structured text searching into object-oriented databases (e.g. [CACS94]), which generally result in expressing the structure as a (hierarchical) network, linked by part-of attributes. Queries are expressed as path expressions in the general language of the database. This approach, although powerful, results in inefficiencies generated by not fully considering the semantics of inclusion (see [CM94] for an excellent discussion on this problem).

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 [SDAMZ94] 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. For example, in [CM94] it is shown that structure-related queries are handled better by a query engine that knows about the semantics of hierarchies than by a general-purpose object-oriented database language.

We rely on this approach. We design a language which is focused on exploiting the 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 [SDAMZ94] for some ideas on this subject.

2.2 Novel Approaches

These approaches are characterized by two important facts.

First, the structure of documents is assumed to be more or less hierarchical; this in part imposes a minimum level of structuring power to a model to be considered acceptable (e.g. the traditional IR model would not qualify), and in part restricts the number of alternatives, by avoiding extremely powerful approaches, as it would be the hypertext model. This restriction seems reasonable, since up to now even this simpler problem has not been solved satisfactorily.

Second, the query language is required to integrate in a uniform syntax the queries on contents and on structure. The structure of text must be stored in some way, for example by *marks* embedded in the same text (SGML, for example [ISO86, Gol90, RTW93]), or by keeping auxiliary indices.

We present a sample of novel models, which cover the different approaches to solve this problem under the stated conditions. See [Loe94] for another survey.

2.2.1 A Simple Model: Flat Structure + Full Text Retrieval

Perhaps one of the simplest approaches is [BY94], which has been partially implemented in SearchCity [Ars92]. The idea is to have a database composed of a set of documents (or files, if no structure is defined), which may have fields. Those fields need not cover all the text of the document, and can nest and overlap. The subject of how to parse a document to get the index points is also considered. Finally, 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 union, intersection, difference and complement of both documents and match points; for restricting matches to only some fields, and to retrieve fields owning some match point. Since it is not possible to determine whether a field is included in another (except under certain assumptions on the hierarchy), and it is not possible to make certain compositions of expressions involving fields, we say that this model is "flat". We present more details on this language later, when we compare expressivity.

This model is more expressive than traditional ones, mixing the best of document retrieval [SM83] and full text retrieval. Although we are not interested in how the indexing is performed nor on the matching sublanguage, the model makes a first incursion on the problem of mixing queries on structure and contents; we take only this aspect into account to situate it in the framework of expressivity. This model can be implemented very efficiently.

2.2.2 A Model of Structuring Based on Matches

Another approach is PAT expressions [ST92], a model that has been implemented in the PAT Text Searching System [Faw89]. Again, there is a lot of concern on the mechanism to index the database, which we are ignoring here. The only index is on match points, there is no indexing on structure. For this purpose, the language allows dynamic definition of structures, based on match point expressions for the beginning and end of *regions*. It also allows to use externally computed regions. Although the dynamic definition approach is flexible, it relies on specific markup requirements: it must be possible to express regions by simple expressions on match points. For example, it does not allow to recognize the structure of "C" code. This idea has been applied successfully to the computerization of the Oxford English Dictionary (the OED project [BGT91, Gon87]), because it uses an SGML-like markup. It is difficult to imagine other type of structuring where this approach works.

Structures can have substructures; this fact is not explicit, but derived from the inclusion relationship between regions. By the form of definition of structures, it follows that recursive structures are not allowed, each structure owns a set of non-overlapping areas of the text.

Although it is not supposed to depend on the underlying implementation of the index, the operations defined on the text are oriented to the use of a PAT array. Indeed, some operations are included mainly because they are easy to implement with a PAT array, although, as it is pointed out in [ST92], they are rarely used and difficult to grasp and even to specify (e.g. `lrep`).

Another characteristic of this language is that it mixes the concept of match point with the concept of region. This distinction is perhaps inherited from the way the PAT array works, and causes a lot of troubles and lack of orthogonality and compositionality in the language, as is pointed out in [ST92] (e.g. `signif`). Sometimes, it is even impossible to determine statically whether the result of an expression is of type "matches" or of type "regions" (e.g. the $+,-$ and `fby` operators).

Despite these drawbacks, the model is a good example of structuring documents and querying them by mixing contents and structure. What is most important, since all operations are based on the PAT array, they are extremely fast. Operations on areas are also fast, thanks to the restrictions imposed on structures: non-overlapping and non-recursive. Finally, the space requirement is low if the structure is dynamically extracted from match expressions on the text. Thus, it achieves high efficiency at the cost of some restrictions, which for some applications are reasonable. This is an example for a model based on a pre-defined index on the text.

2.2.3 A Model Based on Overlapped Lists

Another approach quite similar to PAT expressions but less powerful is [Bur92a, Bur92b], which also has flat lists of disjoint segments, originated by textual searches or by "regions" like chapters, for example. The idea is to unify both searches by using an extension of inverted lists [FBY92, chapter 3], where areas are indexed the same way as words are. The operations are simple: select regions containing or not containing other regions; select regions contained or not contained in other regions; select a given region or a given word; and other operations more close to traditional IR (e.g. relevance ranking).

A recent work extending this idea is [CCB95a, CCB95b], which enhances the algebra with overlapping capabilities, some new operators and a framework for an implementation.

The new operators are: union (*one-of*), *followed-by*, *both-of* and *n-words*. In fact, *both-of* is just a symmetric *followed-by* (i.e. (A *followed-by* B) + (B *followed-by* A)). *n-words* returns all (overlapping) segments containing n words.

An interesting feature of *followed-by* is that it selects the whole segment between the matched operands, not one of them. Another interesting feature is that, although the model does not allow inclusion between regions of the same kind, it allows overlaps. This is obtained for example by the definition of *both-of*, since in case of a text of the form "aba", the query ("a" *both-of* "b") returns two overlapping segments: "ab" and "ba". It is not clear whether this feature is good or not to capture the structural properties that structured information has in practice.

The implementation relies in four primitives, that are used to iterate on the operands to produce the result. Since both the operands and the result are flat lists, the implementation can be very efficient.

2.2.4 A Model Based on Lists of References

In [Mac91, Mac90], a model is proposed to uniform definition and querying of structured databases, by means of a common language. It is strongly based on SGML [ISO86], although not dependent on it.

The language is somewhat outside the scope that we have proposed for similar models, since it does not only include data definition features, but also hypertext-like linkages and some operations closer to object-oriented databases (by means of allowing nodes to have attributes that can also be queried). It is also possible to incorporate "external procedures" to the query language, much as in object-oriented databases.

Although the structure of documents can be hierarchical (with only one strict hierarchy), 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. This allows to integrate in an elegant way answers to queries to hypertext links, since all are seen as lists of references. The model has also some navigational features that allow traversing those lists of references.

The structuring mechanism is also complex, since some storage organization facts are expressed as structure. The top-level of the hierarchy has *archives*, which comprise a set of *groups*. Each *group* have a set of *documents*, *references* and more *groups*. Each *document* has a hierarchy of *elements*. The *references* can be retrieved and treated the same way as the result of a query (which is in fact a list of *references*).

This model is very powerful, and because of this, has efficiency problems in its implementation [Mac91]. To make the model suitable for our comparisons, we consider only the portion related to querying structures. Even this portion is quite powerful. A related, although more navigational model is presented in [DWL92].

2.2.5 A Model Based on Manipulating Parsed Strings

This approach has also been used for the OED project [BGT91], but in different problems [GT87]. Those problems are related to transforming a database, or to generating new views by means of processing the data and structure. It has been successfully applied to the Short OED (SOED) project [BBT92], for example, in which the goal is to extract a shorter version from the original dictionary.

Since it has to be a data manipulation language rather than a plain query language, the approach is quite different. The language used to express database schemas is a grammar (regular, context-free or context-sensitive), 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 (only context-free grammars are considered). The parsing process implicitly comprises the work of pattern-matching, since there are no further operations to express text matching. The language also relies on the facilities of its host language, Gödel, based on Maple [CGG+88].

There are a number of powerful operations that can be performed to manipulate parsed strings: they can be reparsed by another grammar, some nonterminals can be hidden, etc. Since those operations are the usual ones, we can infer that reindexing, if done, must be carried out very efficiently. With those operators, the job of taking into account all the complex variations that appear in the structure of the dictionary is simplified, although not eliminated. There are also querying operations. The approach is extremely powerful, and it is shown to be relationally complete.

The problem is efficiency. Being such a dynamic approach, it is hard to implement it efficiently. Some considerations are made about efficiency, but there are no good solutions yet. In [BBT92], we can see that that operations are really slow, although this was not a concern for the SOED project.

A formalization of a data manipulation model based on grammars, quite similar to this approach, can be found in [GPG89].

2.2.6 A Query Language Based on Tree Matching

In [Kil92, KM93] a model relying on a single primitive, tree inclusion, is proposed. The idea of tree inclusion is, seeing 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 approach is not meant to be comprehensive in expressivity, but to deeply study the properties of that primitive.

The main idea of tree inclusion is that it is a way to query on structural properties in which the user does not need to be aware of all the details of the structure, but only on what he/she is interested. This stands for data independence.

Simple queries return the roots of the matches, and 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.

Although the language is set-oriented, the algorithms work by sequentially obtaining each match. The use of logical variables makes the problem intractable (NP-hard in many cases), and even without them, unordered tree inclusion is NP-complete. Ordered tree inclusion of a pattern P into a textual database T takes $O(|P||T|)$, and $O(|T|)$ if the structure is not recursive (i.e. no node can be the ancestor of an equally labeled node). See [Kil92, KM92, KM95] for this study of complexity.

Finally, the paper suggests that since the leaves of the pattern can be text matching expressions, a good query plan consists of first searching for those text matches, and then trying to match the pattern only on those positions, this way using a bottom-up heuristic rather than a top-down approach. We pursue that idea much further to achieve practical solutions to the problem.

2.3 A Formal Study on Expressivity

In [CM95], a simplification of PAT expressions [ST92] is used to formally analyze its expressive power. It uses a single hierarchy and a set-oriented query language, with set operators for union, intersection and difference, operators to select elements including or included in others, and operators to select elements before or after others.

The paper relates this algebra with monadic tree theory, which provides an alternative point of view on the expressive power of the language. It is found that it is not possible to express direct ancestorship (i.e. ancestorship without intermediate nodes) nor *both-included* in the algebra. *both-included* is a ternary operator defined as follows: *both-included*(A, B, C) iff there is a node A ancestor of nodes B and C such that B is before C. Another restriction of the algebra is that it cannot join on contents, e.g. select chapters whose title appears in a given list of titles (which is also in the text).

The paper shows that by extending the language to manipulate tuples (e.g. it currently can select A nodes including a B node or B nodes included in an A node, but not pairs (A, B) such that A includes B), and allowing joins by identical nodes, the algebra can express direct ancestorship and *both-included*, and suggests that *n-included* cannot be solved with a 1-tuple algebra (since the demonstration that shows the inability to express *both-included*(A, B, C) can be extended to show the inability of this operator to express *three-included*(A, B, C, D), and so on). On the other hand, it shows that allowing joins by text contents make optimization problems much harder.

Observe that the language of [CCB95a] is a 1-tuple algebra and allows to solve the *n-included* problem. *n-included*$(c, a_1, ..., a_n)$ is equivalent to c *including* $(...(a_1$ *followed-by* $a_2)...$ *followed-by* $a_n)$. This is because the *followed-by* operator does not return elements from its operands, but create new segments that includes pairs of elements from its operands. This is a way to "codify" a tuple, but it cannot be used beyond the scope of flat hierarchies, since it would produce overlapping elements. In fact, we will show that our model does not solve the general *n-included* problem, and however includes this feature of [CCB95a].

Returning to [CM95], this paper (and a related one, [CM94]), focuses also on optimization problems. It defines a Region Inclusion Graph, where the nodes are the different labels of the nodes of the hierarchy and the edges represent the possibility of direct ancestorship between nodes labeled that way. It also defines a Region Ordering Graph, representing the possibility of nodes appearing in a given order. Both graphs are used to detect redundant work in the query plans. A study on the complexity of optimization algorithms is also presented.

[CM94] also integrates the presented textual database engine with a classical database, by seeing the textual database as a classical one and translating the classical query operations into the language of the underlying text engine, and applying the optimization techniques. Some operations are much more efficient implemented this way.

Chapter 3

A New Model for Querying Structured Text

In this chapter we present our model. A model must include both the type of structuring allowed for the database, and the language to query on that structure and on the contents. We first point out some considerations about the problem, and then present our model.

- There are different ways to express structure, e.g. SGML markup, grammars for parsing, etc.
- There may be parts (or the whole) of the text with no structure, e.g. a highly structured form may have a place for free annotations, which has no internal structure.
- We are focusing on hierarchical structures, e.g. a book has an introduction and a set of chapters, which may have sections, subsections, etc. which are composed from paragraphs.
- We do not restrict ourselves to strictly hierarchical structures, e.g. the same book may also have volumes, pages and lines, which overlap with the previous structure.
- It is not normally possible to impose *a priori* limits on the complexity of the structure, since it may be recursive, e.g. the structure of "C" code has no limits on nesting.
- The structure may be coarse- or fine-grained, depending on the problem, e.g. a book vs "C" code.
- It is not possible to establish general criteria for the distribution or shape of structures, e.g. declarations on "C" code are concentrated on the beginning, most formulas may be in a single chapter of a book, etc. This means that a meaningful study on the average running times of our algorithms is not possible in general.

All these facts need to be taken into account at the time of designing a structuring model. We present now an informal description of our model, to enhance intuitiveness. We formalize these notions in a later chapter.

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" (e.g. [KM92]), searching what we want across the whole tree of the structure.

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 delete those not meeting the search criterion.

Our solution is a language oriented to sets of nodes. 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 sets of nodes and produce new sets of nodes. Figure 3.1 shows the main concepts, and will be refined along the work, to detail the query language and to draw a general software architecture comprising this model.

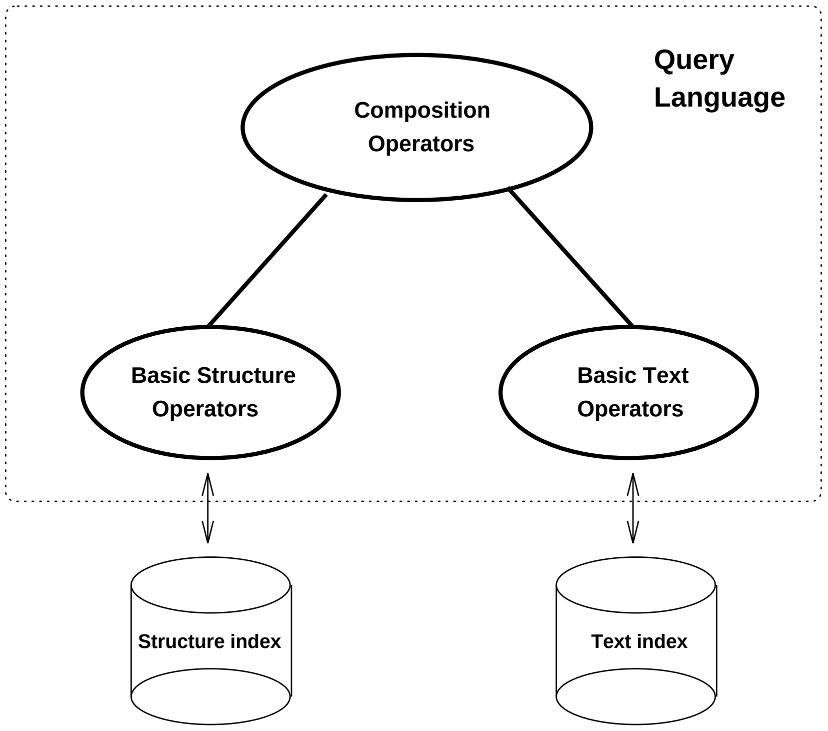


Figure 3.1: Initial diagram of how our model operates.

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 sets 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 operations. Observe that, this way, 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. 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 [KM93, GT87], 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 by using adequate data structures; 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.

Removing 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 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 only 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 (recall the example on chapters and pages, or see [SW87] for an example of what could be a view to index only to presentation structures). Although views are strict hierarchies (no overlaps), they do not have to cover the whole text, some portions may not be reachable through a particular view. 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 needs not to be strict). The correspondence between nodes and segments is important, since (unlike *p-strings* [GT87]) a node cannot be dissociated from its segment.

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 imposition 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 is 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 features that may be useful, in order to select an appropriate subset for each application.

We would like to be able to express

- Pattern-matching expressions on the contents of the text; we prefer to be independent of the pattern-matching sublanguage. It could be as simple as single-word matching or as complex as regular expressions with proximity operators, etc. Appropriate pattern-matching languages already exist.
- Operations on those matches: collapse, intersect or shift segments, seeing the top-level nodes of a query as the result of pattern-matching, etc. These operations are to be considered, together with the previous one, as part of the pattern-matching sublanguage.
- All the nodes which belong to some constructor, e.g. all chapters.
- Things which include or are included in others (segment inclusion), e.g. chapters including a figure, or figures included in a section.
- Things preceding or following others, both included in some other structure, e.g. "computer" preceding "architecture", in the same paragraph.
- Things which are at a given position inside others, e.g. the second paragraph of all chapters.
- Direct ancestors or descendants in the tree, e.g. sections with three or more top-level paragraphs (not included in a subsection).
- Things whose contents are the same than others, e.g. the chapter whose title is "Introduction".
- Union, difference and intersection of queries.

It is still possible to add a number of requirements to the query language, but we have to take care on that the operations proposed operate with nearby nodes. Operations not satisfying this cannot be implemented efficiently (see later a discusion about semijoin).

We define a set-at-a-time algebra, following [ST92, GT87], and to mimic the idea of the relational model, which has shown that a set-oriented language is in general much better than a navigational one. 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. Thus, a query returns a set of nodes of some view, not a subtree, so it is not possible to manipulate subtrees nor use queries as "views" (in the relational sense). In this sense, this approach is weaker than [GT87].

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 presents a strict hierarchy. In the approach of [CCB95a], the other choice is selected, i.e. overlaps are allowed in answers, but not nested components.

Although it is not possible to retrieve subtrees, 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 [KM93].

This language is an operational algebra, not necessarily intended to be accessed by the final user, as the relational algebra is not used 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.

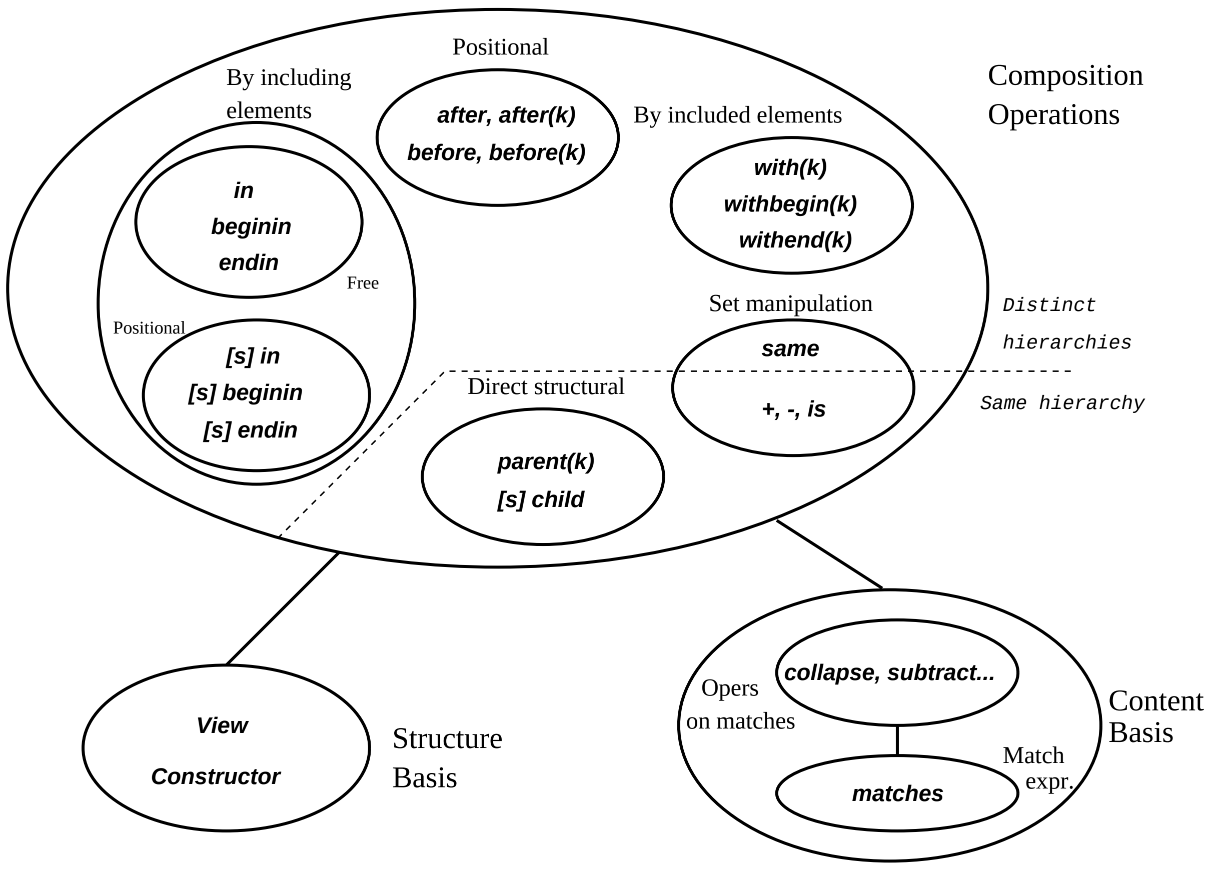


Figure 3.2: The operations of our model, classified by type.

In Figure 3.2 we outline the schema of the operations. There are basic extraction operations (forming the basis of querying on structure and on contents), and there are 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. However, we show a number of good candidates for this set.
  - **M collapse M':** Is the set of segments obtained by superimposing both sets of matches; when an overlap results, they are merged. For example, `"John Smith"` **collapse** `"J. Smith"` gets the apparitions of John Smith in a text, in both forms.
  - **join M:** Pairs of segments of M touching each other are merged. For example, **join** `">*\n"` gets all referenced lines in an e-mail file (that follows that commonplace convention) and packs contiguous referenced line, this way getting referenced paragraphs.
  - **M subtract M':** From the segments of M, we delete the points which are in a segment of M'. Resulting empty segments are also deleted. For example `"Name: *\n"` **subtract** `"Name:"` isolates the names following "Name:".
  - **M intersect M':** Is the set of segments of M, deleting points which are not in a segment of M'. For example, `"Chapter 8 * Chapter 9"` **intersect** `"Page 5 * Page 6"` selects the text of chapter 8 which is in page 6 (assuming that sui generis notation).
  - **complement M:** Is the set of segments comprising all points which are not in a segment of M, making segments as large as possible. For example, **complement** `"^>*\n"` gets the portions of a mail file that do not reference other mail messages.
  - **move(n,m) M (C):** All segments of M are displaced, n positions the left point, m the right one (n and m can be negative). The movement is limited to avoid a segment getting out of the smallest segment of C which previously included it. C is any query from any view (not only textual). For example, **move**$(2, 0)$ `"19??"` (*none*) could get the last two digits of years from this century appearing in the text.
  - **M to M' (C):** Is the set of segments that extend from the start point of a segment of M to the end point of a segment of M', for the closest pairs of segments of M and M'. That means that a segment is in the result only if no segment included in it has qualified. C can be any query (not only textual), and is used to restrict pairs: a pair (m, m') can be selected only if the minimal segment of C including m and m' is the same or does not exist in both cases. For example, `"<section>"` **to** `"</section>"` (`chapter`) selects inner sections in a SGML-like tagged document. This allows a restricted version of *both-included*, as we see later. Observe that this operator cannot produce overlaps, since if two segments (a_1, b_1) and (a_2, b_2) are generated such that $a_1 < a_2 \leq b_1 < b_2$, then none of them should have been generated, (a_2, b_1) should.
  - **toplevel P:** Is the set of segments comprising all points which are in a segment of P (note that P is any expression, not only of matches), making segments as large as possible. For example, one can get the text of chapters without figures with (**toplevel** `chapter`) **subtract** (**toplevel** `figure`). This can be done only by seeing chapters and figures as text segments.
  - **beginnings/ends P:** Is the set of all points beginning/ending some segment of P. Each resulting point is considered a separate segment. For example, one may want to say **beginnings/ends** `italics` to know where to send the appropriate commands to the printer.

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.

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 included other paragraphs, only the top-level ones would be considered, and those partially included in a page would be 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 which extend for 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, 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, as we see in the implementation.

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. This appears to solve the problem of *both-included*, but it does not, as we see later. 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 the 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 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 segment 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.

Note that the only moment in which we access the text is in pattern-matching subqueries, which are leaves of the syntax tree of the query expression, and that names of nodes are converted to the set of their segments. From then on, the rest of the operations manipulate nodes based on their identity and their segment, not text contents nor structures. By only manipulating nodes with a suitable data structure to combine them and to easily obtain the set of segments of a given constructor, a quite expressive language can be obtained, which is at the same time efficiently implementable, since we know from the start where to combine the elements for matching structures.

At this point, we could consider a "semijoin" operation (since the full join needs tuples and is completely outside the style of the algebra). A semijoin selects from one set the elements that would participate in a join with other set. For example, suppose that we have an operator P **says** Q (where P and Q are from the text view), which selects the elements of P whose text is equal to the text of some element of Q. This is not like **same**, that compares segments, in this case we compare contents.

A constructor like this one would allow us to express queries like "give me the books that are referenced in this list" (think of hypertext, for example).

Many implementations are possible. For example, we could read all segments of Q from the text file (in a single pass, if we have the set sorted by segment position), build an automaton [HU79] to recognize any of the strings, and then passing the automaton on the segments of P (another single pass on the text), selecting those segments where the automaton recognizes exactly the segment. This solution is time-efficient, but has to store all the strings of Q.

Another possibility is to store hash values of the strings, and then traversing P doing the same. If we find that the hash value of a P segment is equal that of a Q segment, we read again the Q segment to make the exact comparison. As it can be seen, less space but more seeks are needed.

Still a third possibility is to use a PAT array to search for each Q segment. Each search returns an area of the PAT array with all the points of the database where the text begin like the Q segment. Then, we search each P segment into the set of all match points returned by the PAT, selecting those which are found. Observe that we only have to read the contents of Q segments, not P segments. This approach is $O(n \log_2 T)$ time (where n is the size of the operands and T is the length of the text of the database) and makes no seeks, although it may require to store a lot of match points (a word each) if the words of Q are too frequent. We can store the areas of the PAT array instead of each of their segments, making the algorithm $O(n)$ space, but then each P element has to be sequentially searched in each area, thus adding $n \times$ *matches* to the time complexity.

The problems are many. First, it is no longer true that the text is only accessed at the leaves of the query syntax tree. This, on the one hand, does not help to isolate the text search engine from the rest of the system, since we have to access the database from our own. On the other hand, can make text management much more inefficient, since a good text search engine can answer the normal questions about text with little access to the real database, by using indexing.

It is because of this that we prefer to have an algebra oriented to node proximity. Our point is that that algebra is much more efficient if we do not have operators like semijoin. Although we lose some expressivity by eliminating it, we prefer to sacrifice that expressivity in terms of efficiency and purity of the model.

3.3.2 Examples

We present some interesting examples of the use of these operators.

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** `"earth") (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 example is illustrated in Figure 3.3.
- `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.

- `VP with (Q)` solves the problem of [SW87], by giving the sequence of presentation commands to follow in order to present matches from query Q. Those commands are obtained by traversing the tree in depth-first order.
- `[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.

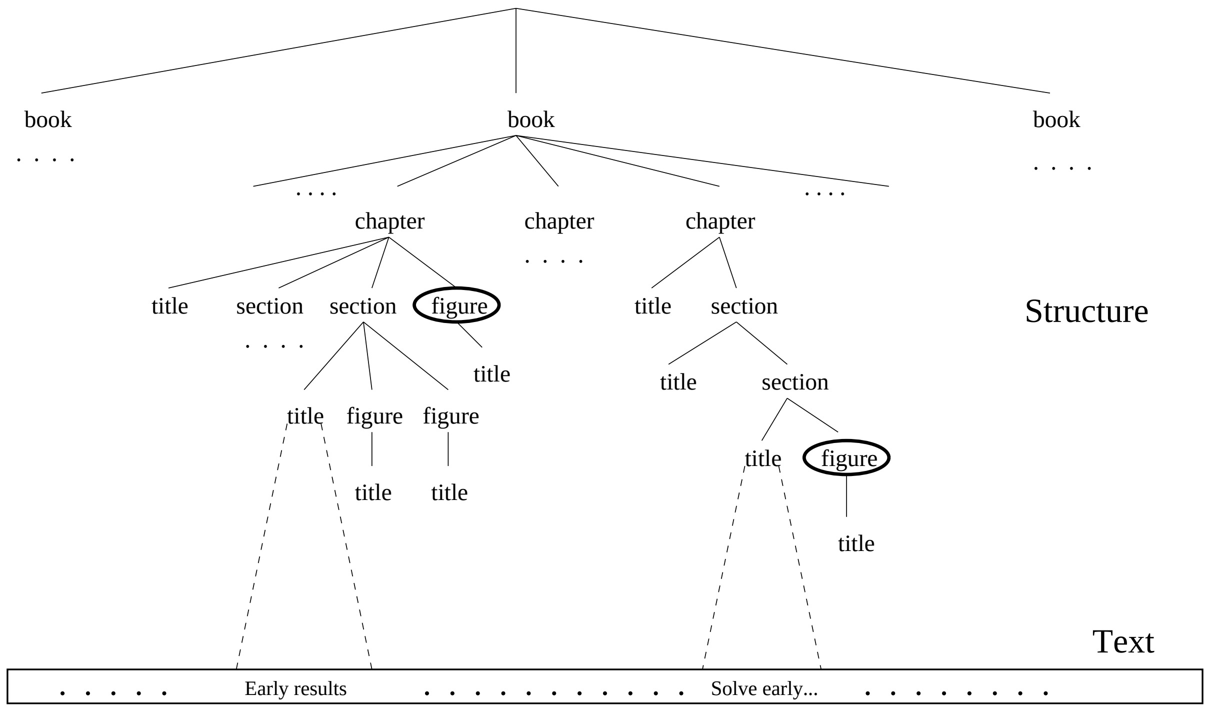


Figure 3.3: Illustration of the effect of the query `[last] figure in (chapter with (section with (title with "early")))`. Bold circles are selected nodes.

3.4 A Software Architecture

In this section we outline a possible software architecture for a system based on our model. Later, we show how the prototype follows these lines.

Our language is not intended to be accessed by final users. Rather, it presents an operational algebra onto which a user-oriented query language can be mapped.

Users should interact with our system via an interface, in which they define what they want in a friendly language (see [KM93] 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.

We pose some requirements to the text engine. The minimum that we need is a subsystem offering us a pattern matching language, to which we can submit a query in that language and it returns a set of (non-nested and non-overlapped) segments of the text that match the query. The subsystem is the only responsible for the view we have on the text, and which can perform markup filtering, stopword elimination, synonym substitution, use a thesaurus, etc. We access the text only through this engine, which is also responsible for indexing the text for its operation. Finally, it must provide a means to retrieve filtered text upon submission of a segment. Notice that if the text subsystem makes any kind of filtering or if it partitions documents into multiple files, it is responsible for keeping consistency between positions in the filtered file (as the upper layer sees it) and positions in the real file. These positions can be word positions, character positions, etc.

See Figure 3.4 for a diagram of how a complete system based on this schema should be. The figure does not illustrate the indexing process. The “document layer” is intended to support more sophisticated document management, such as collections of documents, etc. It translates all references to collections (as the user sees them) to references to documents (as they are indexed in lower levels).

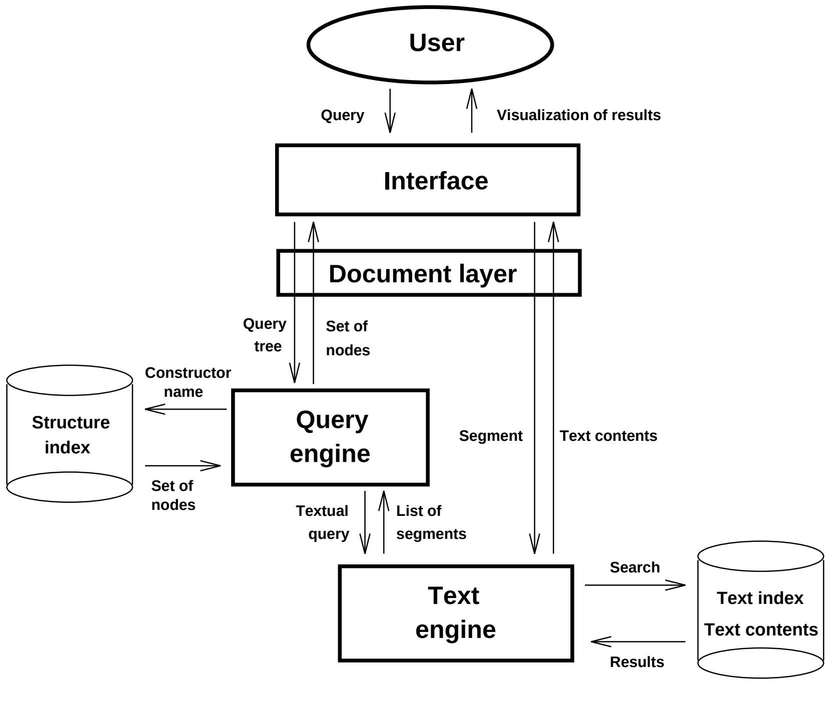


Figure 3.4: The architecture of a system following our model, regarding querying. Arrows illustrate the process of solving a query.

The indexing process is presented in Figure 3.5. 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.

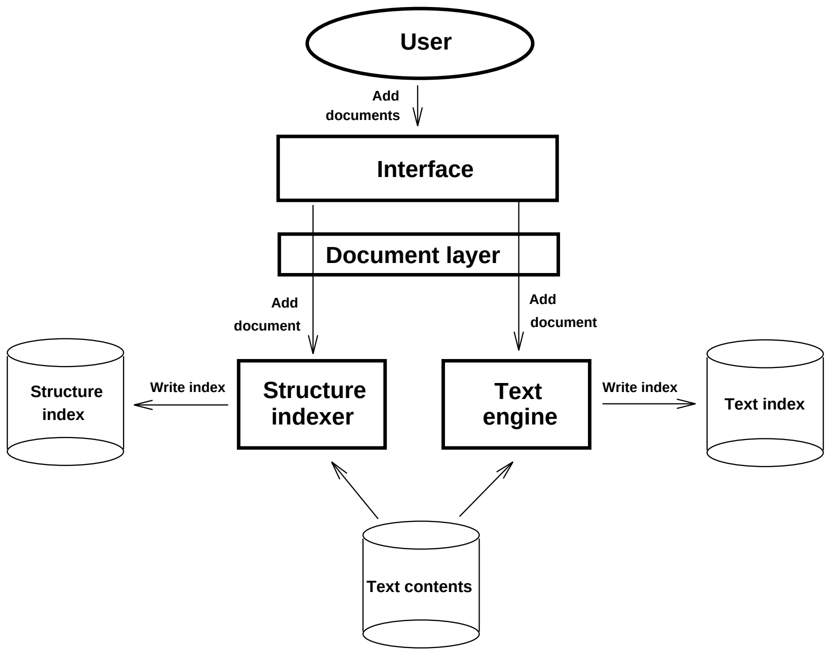


Figure 3.5: The architecture of a system following our model, regarding indexing. Arrows illustrate the steps to add a document to the database.

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.

Chapter 4

Expressivity

The aim of this chapter is to study the expressivity of our model. We begin with a formalization of the description of Chapter 3, then we formally compare our model against each one of the similar models and finally we draw an informal framework to situate the expressivity of any similar model.

We are not going to treat our proposal as a complete language. The idea is to make an operator-wise comparison, i.e. which operators we need to represent each of the similar models, and vice versa.

4.1 Formal Model

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

- $\Sigma = I(\mathcal{T})$ is the alphabet of the text ($I(f)$ denotes the image of function f).
- $\mathcal{T} : [1..T] \rightarrow \Sigma$ is the text array. T is the size of the database (number of symbols).
- $\mathcal{V}$ is the finite set of views over the text, with a distinguished element $V_t \in \mathcal{V}$ (the text view).
- $\mathcal{C} = I(C)$ is the finite set of constructors, with a distinguished element $C_t \in \mathcal{C}$ (the text constructor).
- $\mathcal{N} = I(N)$ 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).
- $C : \mathcal{V} \longrightarrow \wp(\mathcal{C})$ is the set of constructors of each view, we also write $C(V)$ as C_V. It holds $\forall V_1 \neq V_2 \in \mathcal{V}, C_{V_1} \cap C_{V_2} = \emptyset$. Also, $C_{V_t} = \{C_t\}$.
- $N : \mathcal{V} \longrightarrow \wp(\mathcal{N})$ is the set of nodes of each view, we also write $N(V)$ as N_V. It holds $\forall V_1 \neq V_2 \in \mathcal{V}, N_{V_1} \cap N_{V_2} = \emptyset$. Also, $N_{V_t} = \{t_{a,b} / 1 \leq a \leq b \leq T\}$.
- $R : \mathcal{V} \longrightarrow \wp(\mathcal{N} \times \mathcal{N})$ is the binary relationship which defines the tree of each view, we also write $R(V)$ as R_V. It holds $\forall V \in \mathcal{V}, R_V \subseteq (N_V \times N_V)$. Also, $R(V_t) = \emptyset$.
- $Constr : \mathcal{N} \longrightarrow \mathcal{C}$ is the constructor of each node. It holds $\forall V \in \mathcal{V}, \forall x \in N_V, Constr(x) \in C_V$. This implies that $\forall a, b / 1 \leq a \leq b \leq T, Constr(t_{a,b}) = C_t$.
- $Segm : \mathcal{N} \longrightarrow [1..T] \times [1..T]$ 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 $\longrightarrow$.
- $\forall V \in \mathcal{V}, \exists!\ r_V \in N_V /\ \nexists x \in N_V / x \longrightarrow r_V$, that is, each view 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.
- $\forall V \in \mathcal{V} - \{V_t\}, \forall x, y \in N_V, Segm(x) \subset Segm(y) \Rightarrow x \longrightarrow^{+} y$, that is, except in the text view, 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 view, 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 view).

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

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. In Appendix A we define *Expr* (our language syntax) by an annotated abstract syntax, and then define $\mathcal{I}$. With that formal definition, we can compare our model against others.

It is important to note that our model includes all operators proposed in [CM95] and more, so its positive results about expressivity apply also to us, while some negative results do not (e.g. we can express direct ancestorship, but not *both-included* nor joins on contents).

4.2 Comparison with Similar Models

In this section we compare our model against those which are close to it, namely the novel approaches.

We compare first our model against the hybrid model [BY94], PAT expressions [ST92], overlapped lists [CCB95a], lists of references [Mac91] and the tree matching model [KM93]. We determine which aspects of one model can be represented in another one.

The *p-strings* model [GT87] lies outside this comparison, since it is a data manipulation rather than a query language. We can say, intuitively, that it is very expressive. Recall also that we compare only the portion of lists of references [Mac91] related to querying structures.

Before entering into the details of each case, we define here the *overhead* of a representation, as the ratio between the translated query length over the original one. This is defined as if only one translation rule were to be applied, for example if a translation rule duplicates an operand, we say that the overhead is 2, although a formula with n operators like that one will be 2^n times longer when translated.

4.2.1 The Hybrid Model

We show that our model subsumes this simple model. Some observations follow.

This model consists of documents, fields and match points. Fields can overlap freely. Thus we have a view of documents (called DV) and a number of views for fields, as many as necessary to avoid overlapping nodes inside a single view. We represent a field f by the constructors $f_1..f_r$, where $r > 1$ if we need to split the view of that field because of overlaps.

Answers in the model are pairs (D, M), where D is a set of documents and M a set of match points, so we represent each query of this model with two of ours, one for D and another for M.

Finally, we do not compare the matching sublanguage, since our interest is not there.

We define two functions, $\mathcal{D}$ to represent the D component and $\mathcal{M}$ to represent the M component. Always $\mathcal{D}() \subseteq DV$ and $\mathcal{M}() \subseteq V_t$ (in fact, size-1 segments). For a formal definition of the semantics of the hybrid model, see [BY94]. The definition of the $\mathcal{D}$ and $\mathcal{M}$ functions is depicted in Appendix B.1.

From the definition follows that we can represent all the model, although with some overhead: $O(m2^m)$ if $some(q_1..q_m)$ is involved, $\max(2, r)$ otherwise.

The converse of this representation is weak. The hybrid model can represent little from ours, because of its poor structuring power. It can define all our hierarchies, but without obtaining information about ancestorship.

Suppose we have a single document comprising all the database, and all the constructors from all views are represented as fields (we cannot use documents because they are too strict, thus we can only use match points). We represent the beginnings of the segments, so suppose there are no two segments with the same starting point (which is a restriction). We use a function $\mathcal{R}$ to denote the translation, which is defined in Appendix B.1.

As the definition of the function shows, little can be represented in this model. There is no overhead in the subset that can be represented.

4.2.2 PAT Expressions

This approach does not have a very powerful structure, albeit it has good pattern matching primitives. We begin by showing that we can represent this model almost completely (disregarding text matching and some undesirable complications).

Since each region definition is a set of non-overlapping (and, in fact, non-nesting) segments, we use a view V_{reg} for each region reg, and our answers are text segments with the resulting areas. The language is quite complicated with respect to the conversion between regions and match points, and this is in fact not good, so we disregard this subject as long as we can, assuming we deal with regions ($\approx$ segments). In any place where a conversion from regions to match points is needed, we can apply the **beginnings** operator. This conversion cannot be syntactically determined, so we cannot represent it. However, these conversions are more a problem of the model than a powerful feature we would like to represent in ours.

We define a function $\mathcal{I}$ to represent the translation. For a description of the semantics of the language, see [ST92]. The $\mathcal{I}$ function is defined in Appendix B.2.

As it can be seen from the definition of the function, most of the model can be represented in our terms. The overhead is no more than 2.

Now the converse. A strong restriction is that our views have to be flat. We suppose we have a region reg_c for each constructor c, and we return segments from regions. We use a function $\mathcal{P}$ to denote the translation.

$\mathcal{P}$ is defined in Appendix B.2. From its definition follows that we have to lose a lot of structuring power to translate our model into this one. The subset that can be translated has no overhead.

4.2.3 The Overlapped Lists Model

The structuring mechanism of this approach is quite similar to PAT Expressions, so the comparison is analogous. We show that, except for overlapping regions, our model subsumes this one.

We use views V_{reg} and the rest as before. We cannot represent overlapping regions, so this is a restriction. For a description of the semantics of the language, see [CCB95a]. The $\mathcal{H}$ function representing the translation is defined in Appendix B.3. Since we cannot represent overlaps, we return the **collapse**d version of the original answer.

As it can be seen from the definition of the function, all the model except the overlapping feature can be represented in our terms. The overhead is no more than 2.

Now the converse. A strong restriction is that answers are flat (although they can overlap). We suppose we have a region reg_c for each constructor c, and we return segments from regions. We use a function $\mathcal{W}$ to denote the translation.

$\mathcal{W}$ is defined in Appendix B.3. From its definition follows that, if we disregard the problem of flat answers (which is important), a significant part of the query operations can be translated. The overhead is no more than 2, except for **with**(k), where the overhead is k (i.e. the overhead is exponential, because the operand is repeated k times, while the original formula needs $\log k$ bits to represent k).

4.2.4 The Lists-of-References Model

This model allows the structures to be hierarchical, but it does not retrieve sets with nested elements or sets with elements from different constructors.

We initially discuss how to represent this model. It has just one (strict) hierarchy, so we have a single view V. We define a function $\mathcal{X}$ to represent the translation. For a description of the semantics of the language, see [Mac91]. The $\mathcal{X}$ function is defined in Appendix B.4.

As it can be seen from the definition of the function, all the (selected portion of the) model can be represented in our terms. The overhead is no more than 5 (because of **not having all** n).

Now the converse. A strong restriction is that only one hierarchy can be represented, and that the answers have to be flat and from the same constructor, so only the top-level of our answers can be represented, and unions can be done only in the same constructor. We use a function $\mathcal{Y}$ to denote the translation.

$\mathcal{Y}$ is defined in Appendix B.4. From its definition follows that an interesting part of our model can be expressed with this one, if we disregard the mentioned limitations. The subset that can be translated has no overhead.

4.2.5 The Tree Matching Model

This model is quite expressive in terms of structure, but weak with respect to the relationship between structure and text.

We begin analyzing how can we represent this model. It has just one (strict) hierarchy, so we have a single view. To represent a single leaf c we use **Constr**(c). To represent $constr(T_1, ..., T_k)$ (i.e. a tree labeled $constr$ with subtrees $T_1, ..., T_k$), we could in principle use (...((**Constr**($constr$) **with** T_1) **with** T_2)... **with** T_k), but that is not exactly the same. The reason is that, as [KM93] specifies, a structural property (e.g. ancestorship) must hold in the target if and only if it holds in the pattern, so our query translating $a(b, c)$ (i.e. (a **with** b) **with** c) could return the root of a tree $a(b(c))$, in which holds that c descends from b, which is not true in the pattern. We have not been able to express this restriction in our language. We can use **after/before** to express that they should not descend from one another, but we cannot say that both of them should descend from a (i.e. a **with** (b **before** c ($none$)) will retrieve a from the target $d(a(b), c)$, which is not desired).

This problem is exactly the *n-included* problem mentioned when we surveyed [CM95], which we are confident that cannot be solved with a 1-tuple algebra like ours. Observe that the source of this problem is that it is not possible to express that a constructor appearing in two parts of an expression should denote the same node, and that is exactly what an equijoin would do. We could add a ternary operator *both-included*(A, B, C) to solve the $a(b, c)$ problem, but we could not solve the $a(b, c, d)$ problem, and so on.

By looking at our representation of the overlapped lists model, we can attempt to solve the problem for a restricted case. Suppose we translate $a(b_1, ..., b_n)$ as (a **with** ((**toplevel** b_1) **to** ... **to** (**toplevel** b_n) ($none$)...($none$))). It works if there are no solutions at a deeper level, for example $a(b, c)$ would not be found in $a(b(b, c))$. But it works for certain restricted cases, namely when for each of the b_i there is a b_j such that b_i cannot include b_j (we can achieve this by partitioning the constructor sets). With this restriction we can represent tree patterns, by recursively applying the translation to each of the b_i subtrees. Observe that the overlapped lists model handles this situation more easily.

Before referring to logical variables, which make this model much more powerful but more intractable from the point of view of the algorithms, we should consider that tree pattern-matching alone, as defined, cannot express our inclusion semantics, except by a union of a combinatorial number of alternatives. That means that each model has a different conception of how an inclusion pattern should be understood. While this model interprets that an ancestorship relation should hold in the target *if and only if* it holds in the pattern, our model interprets that the relation should hold in the target *if* it holds in the pattern, but more relations can hold. The subject of why cannot we represent the alternative semantics or how can logical variables be used to reasonably represent our semantics should not distract us from the fact that both models have a basically different conception of what an inclusion pattern is. It is hard to decide which conception is better in practice.

On the other hand, the model has logical variables, which we cannot express at all. They are used to extract other parts of the matching subtree, not only the root (we can do this with other mechanisms implicit in our language, since we express the context to search and at the same time what node we want from the context); and to express that two parts of the match must be identical (what we cannot express at all).

We now consider what in our model can be expressed by the tree matching model. A first restriction is that there is only one view. We use the Prolog-like syntax of [KM93], and a function $\mathcal{K}_X$ to denote the translation, which leaves the result in the Prolog variable X. Suppose the view is called V and the constructors are $c_1, ..., c_r$. The function $\mathcal{K}_X$ is defined in Appendix B.5.

As it can be seen, an interesting part of our model can be expressed with tree pattern matching, being the weakest part those operators related with the textual contents of the database. The overhead is exponential, because of the **with**(k) operator, where the overhead is k (and we need $\log k$ bits to represent k). With the exception of this operator, there is no overhead.

4.3 A Comparison Framework

Table 4.1 summarizes the primitives offered by each model to deal with the many aspects of a query language for structured text. We do not include considerations about the matching sublanguage, which is another important subtopic in practice.

We situate the similar models in a common framework, and use this framework to compare the distinguishing features of each model.

Finding a model of expressivity as it could be the hierarchy of grammars in formal languages [HU79] is certainly an ambitious goal (a first step in this direction could be [CM95]). We content ourselves with pointing out a number of aspects in which (at least) a model should be examined in order to analyze its expressivity. Later, we draw an informal graphical representation of the expressive power of these languages.

In [SDAMZ94], 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 $\langle doc \rangle$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 $\langle doc \rangle$s with $\langle title \rangle$ containing 'parallel' and 'processing' " can be expressed as `doc` **parent** ((`title` **with** "parallel") **with** "processing"). The other

Operation	Set manipulation	Inclusion	Distances	Other features
Our model	+,−, **is**,**same**	**with***(k), ***in** **parent**(k), [s] **child** [s] ***in**	**after**[(k)](C) **before**[(k)](C)	powerful in both aspects: structure and contents.
Hybrid model [BY94]	+,−,&,~ *or*, *butnot*, *and*, *not*	field *with* *in* field	only in matches	both documental and textual queries.
PAT expressions [ST92]	+,−,$\wedge$ not *op*	*including.n* *within*	*fby*,*near*	powerful matching language; dynamic regions definition.
Overlapped lists [CCB95a]	$\bigtriangledown$	$\triangleright$, $\not\triangleright$ $\triangleleft$, $\not\triangleleft$	*n words* $\diamond$, $\triangle$	combination operators and overlaps
Reference lists [Mac91]	**union** **intersection** **difference**	**of**, **from** **in** **having any** k	none	also hypertext and attribute management
Tree matching [KM93]	, ;	tree patterns + variables	none	powerful structural queries.

Table 4.1: A comparison of the operators of the novel models.

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 `View` **child** `article`.
  - "Find elements with children" can be expressed as `View` **parent** `View`.
  - "Find elements where the first child is ⟨*title*⟩" can be expressed as `View` **parent** ([1] `title` **child** `View`).
  - "Find elements within a ⟨*section*⟩" can be expressed as `View` **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` + `book`) with ((`title` **with** "parallel") **with** "computer"). This issue is more concerned with the problem of having the different constructors 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 ⟨*corres*⟩s with attribute 'confidential' = yes". If we have those attributes as constructors children of the node and their values in the text, we can answer simple queries, in this case we express it as `corres` **parent** (`condifential` **same** "yes").

Another attempt to classify these kind of models is made in [Loe94], which surveys a number of approaches to structured text retrieval.

4.3.1 A Methodology to Analyze a Language

We want to make a stricter analysis that the one done in [SDAMZ94], since its requirements are fulfilled by not-so-powerful languages also. We divide our analysis in three main areas.

Structuring mechanism: It refers to the capabilities of the language to express the structure of a textual database. Some questions one should ask here are:

- Is it possible to express a hierarchy? (e.g. some models impose a flat structure).
- Is there any limit on the hierarchy? (e.g. maximum depth, recursiveness, etc.).
- Is the hierarchy strict or does it allow overlaps? (e.g. we allow overlaps only between different views).
- Does it allow to express multiple hierarchies? (e.g. the tree matching model has only one hierarchy).
- Is there any limitation on the construction of the hierarchy? (e.g. PAT expressions has some parsing constraints that impose a nonrecursive structure).

Query language for contents: It refers to the part of the query language related to the text of the database, and especially the way to relate it to structure. Some important questions are:

- How is the string matching sublanguage? (e.g. wildcards, proximity, matching with errors, ranges, regular expressions, etc.).
- How is a matching subquery inserted in the context of a structural query? (e.g. we see text queries as part of a special text view).
- How can restrictions on distances be expressed? (e.g. **after/before**).
- How is the text seen in the model? (e.g. we associate nodes with their segments, other models see almost only text, and others almost do not see the text).
- Is it possible to express relationships between the contents of different parts of the text? (e.g. chapters whose title is listed elsewhere in the text).
- How are the set manipulation features with respect to contents? (e.g. we use different operators, as also does the hybrid model).
- Are text segments first-class objects? (e.g. some models do not allow to retrieve text segments, or to test if a text segment includes something).

Query language for structure: It refers to the part of the query language related to the structure of the database. Important questions are:

- How can ancestorship/descendantship be expressed? (e.g. **in**, **with**).
- Can it distinguish between direct and transitive relations? (e.g. the tree matching model cannot differentiate a child from a descendant).
- Can it discriminate ordering or positions among siblings? (e.g. [s] **in**).
- Can it express relationships between the structure of different parts of a matching tree? (e.g. the tree matching model can select chapters having two sections with identical structure).
- How are the set manipulation features with respect to structure? (e.g. union, difference, intersection, complement, etc.).

Area	Structuring mechanism	Contents query language	Structural query language
Our model	A set of disjoint strict trees (views), with no more restrictions. Views can overlap.	Text is a special view. Nodes cannot be dissociated from segments. Text queries are leaves of query syntax trees. There are powerful distance operators. Text content is accessed only in matching subqueries, thereafter it is seen just as segments. There are special set operators for text.	Can express inclusion, positions, direct and transitive relations; discriminates ordering (with restrictions) and manipulates sets. Cannot express relationships between different parts of the structure. Can express complex context conditions, but not everything.
Hybrid model [BY94]	IR-like documents + fields + text. Fields can nest and overlap, but it is a flat model.	Query = matches + documents. Almost all the language is oriented to matches, which are seen as their start point. Expresses distances. Has separate set manipulation tools for matches and documents.	Only to restrict matching points to be in a given field or to select fields including matching points (selected fields are then seen as matching points). Weak in general.
PAT expressions [ST92]	Dynamic definition of regions, by pattern matching. Each region is a flat list of disjoint segments.	Powerful matching language. Has matching points and regions. Regions are just segments. Has set manipulation operations. Expresses distances.	Weak, since structures are flat. Can express inclusion, set manipulation and little more.
Overlapped lists [CCB95a]	A set of regions, each one a flat list of possibly overlapping segments.	Not specified. Words and regions are seen in a uniform way, by an inverted list metaphor.	Results are flat, although they can overlap. Can express inclusion, union and combinations ($\diamond, \triangle$).
Reference lists [Mac91]	A single hierarchy with attributes in nodes and hypertext links.	Text queries can only be used to restrict other queries.	Results are flat and from the same constructor. Can express inclusions, complex context conditions and set manipulation.
Tree matching [KM93]	A single tree, with strict hierarchy. No more restrictions.	Not specified, orthogonal to the model. Apparently it can only be used to restrict sets of nodes of the tree. Weak link between contents and structure.	Powerful tree pattern matching language. Can distinguish order but not positions nor direct relationships. Can express equality between different parts of a structure, by using logical variables. Set manipulation features via logical connectives.

Table 4.2: An analysis of similar models.

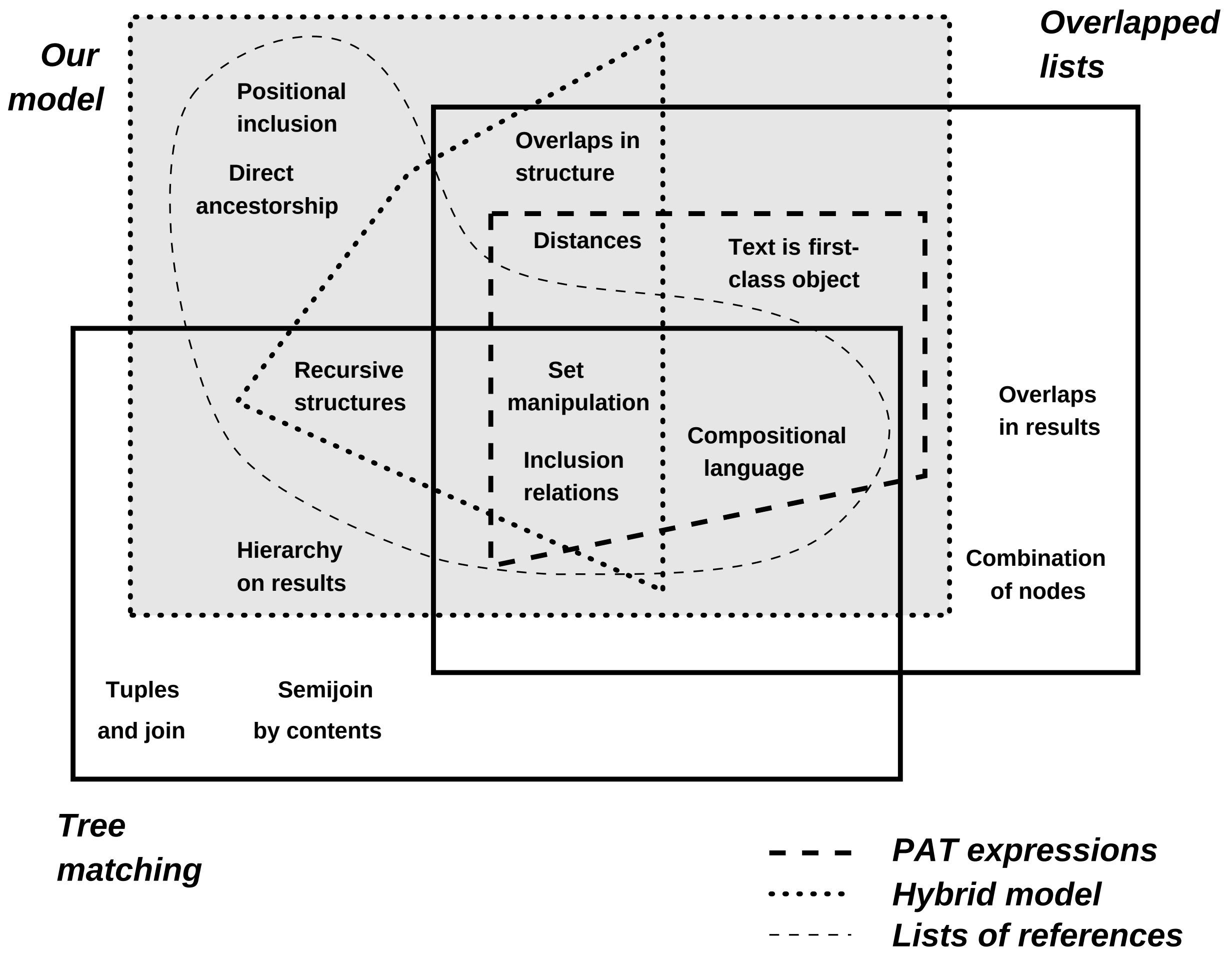


Figure 4.1: A graphical representation of the comparison made in the framework.

4.3.2 A Brief Analysis

Table 4.2 informally compares the similar models (including ours) with regards to the structuring mechanism, and the query language for contents and for structure. We disregard matching sublanguages in this analysis.

In Figure 4.1 we present a graphical version of the analysis done for similar models in the framework. The main desirable features are presented, and each model is represented as a set of the features it supports. Recall that we only consider part of the lists-of-references model.

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.

Regarding tuples, joins and semijoins, only the tree matching model can manage these features (and also *p-strings*, in its own context of a data manipulation language). These two languages have not an efficient implementation. On the other hand, overlaps and combination of resulting nodes from a query are allowed by the overlapped lists model, but at the expense of not allowing them to form a hierarchy. We have not found an efficient implementation if we allow both features at the same time, and consider that the hierarchy is more important in real cases.

This way, we have that our model has most of the features that are important in practice. Those which are not present are not suitable of efficient implementation (although some of them may be important in practice, e.g. tuples and semijoin). We show later that the efficiency of this model is similar to those of less expressivity.

Finally, we draw an intuitive graphical representation of the expressive power of a language, in which we situate the analyzed approaches. It consists of a three-dimensional space, representing more or less expressive power in the three areas: structuring power (z axis), querying on contents (y axis) and querying on structure (x axis). The representation is depicted in Figure 4.2. The placement of each model is done by considering mainly Figure 4.1, but being a quantization of concepts, it is, to a certain extent, subjective. It must be seen just as an alternative description of the results of this analysis, easier to interpret but less formal.

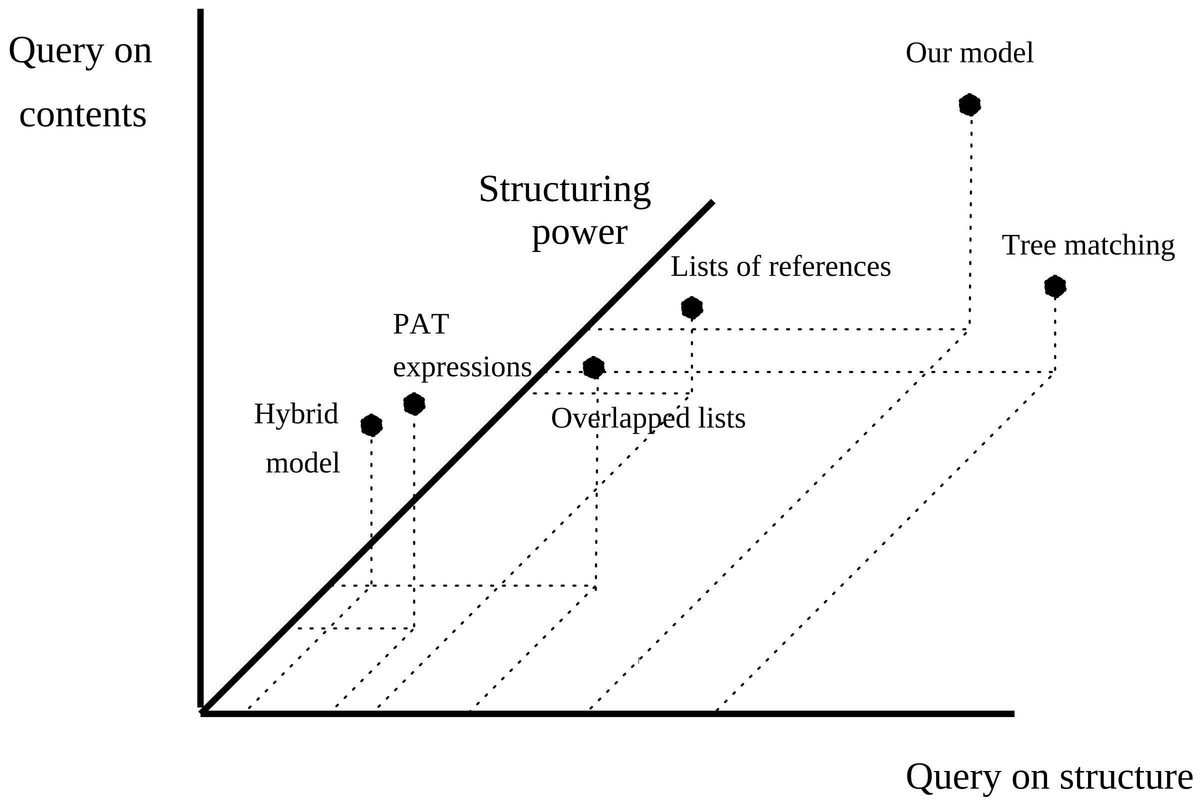


Figure 4.2: A graphical representation of the expressive power of similar models.

4.3.3 Common Limits

From the analysis of a number of structuring models, it also emerges a set of commonplace limits in their expressivity. Although the formal study of these properties is still at the beginning (see [CM95] for some results), we should point out a number of topics that limit most text retrieval query languages. We believe that further improvements on expressivity must begin by considering these features, and how to implement them efficiently.

- Most languages cannot express tuples of values, nor join on their attributes. They are 1-tuple-valued, i.e. any answer from a query is a set of objects, never pairs. Exceptions to this are the *p-strings* model and tree matching (via logical variables). Notice that both of them are quite inefficient to evaluate queries.
- Most languages cannot do a semijoin on text values. We have decided also to avoid it (as we explain in the definition of our language). Again, the exceptions are *p-strings* and tree matching. Also the

object-oriented part of the lists-of-references model can deal with attributes, that can address similar problems.

- The languages that do not have an explicit hierarchy cannot answer direct ancestorship questions, such as finding the immediate parents of a set of nodes. Our model can do this, as *p-strings* does. In this case, tree matching avoids it to encourage data independence (i.e. to avoid forcing the user to be aware of the full structuring details).

- Finally, it is not possible to express *both-included* in most languages (see Section 2.3). The possibility to express *n-included* is related to tuples, and hence can be expressed in the *p-strings* and the tree matching model. An interesting exception is the overlapped list model, which does not allow tuples but allows combination operators, that together with a flat hierarchy with overlaps, allows to express *n-included*. Our model can express it only in restricted cases.

Chapter 5

Algorithms and Associated Data Structures

In this chapter we explain the mechanisms to implement indexing and querying. We present a data structure that allows us to efficiently implement all the defined operations, and many others which are still to be devised. We analyze our implementation in both time and space.

Since our language is fully compositional, we are faced with the problem of, given the syntax tree of a query, compute the set resulting from it. This must be done by recursively solving the subqueries, this way obtaining the resulting sets, which are the operators of the operand at the root. Each operator implements a manipulation of its operands (sets of nodes) to obtain a new set.

A first concern is whether to use full or lazy evaluation. Full evaluation means completely computing the operands before applying an operation, while lazy evaluation means trying to obtain result of the top-level operation element by element, where the extraction of each element triggers the need to obtain certain elements from the subtrees. This way, only the needed part of the operands is computed, which can be much less than the total [GT87].

Consider, for example, the query "`section with(10) "Computer"`". If, instead of first computing the whole tree of sections and subsections and then deleting sections not containing ten "Computer" words, we first compute the top-level sections, see which of them contain ten "Computer" words, and then follow expanding only the subsections of sections that classify, we avoid expanding a lot of unnecesary nodes.

On the other hand, lazy evaluation may force to use less efficient algorithms for some operations, thus it is not immediate which one is better. This has to be experimentally tested.

For full evaluation, we propose two different techniques, which we name *search-operate* and *merge*. The last technique is modified to use lazy evaluation, sometimes losing efficiency as explained.

On the other hand, we can use different kinds of indices, each of them supporting some operations at the cost of some space requirements.

We begin by describing the indexing mechanisms, and then the algorithms for implementing the operations.

5.1 Indexing

We are not going to address the problem of generating the index, since it can be constructed from many different sources. In the prototype, we provide mechanisms for generating the index from a parse tree (which is much simpler to generate), this way allowing any method which generates a parse tree from the source text, to generate an index this way. This mechanism is quite general, since any hierarchical structuring of the text is naturally represented and obtained as a parse tree.

We need indices for two different tasks: for the matching language and for the structure language. We are not going to innovate on the first one. Since it is an independent sublanguage, we also leave open its implementation. For our prototype, we use a matching sublanguage implemented by using a PAT or suffix array [FBY92, chapter 5] and [MM90], but any matching sublanguage and any implementation of it can be used. Indeed, our first prototype used a sequential search engine, without indices.

5.1.1 A Full Index

For efficiently answering queries on structure, we are going to implement each view as a general tree, which embodies all its nodes. As we explain later, the sets of nodes resulting from query evaluation are represented as trees too.

The tree has enough information to extract the nodes of each constructor in time proportional to the number of nodes of the constructor, not the number of nodes of the whole view. That is, each node has pointers to its first child and next sibling in the tree of the nodes of its constructor (see Figure 5.1). Also, the whole view can be transformed into the format of the result of a query in time proportional to its size, by a simple recursive algorithm.

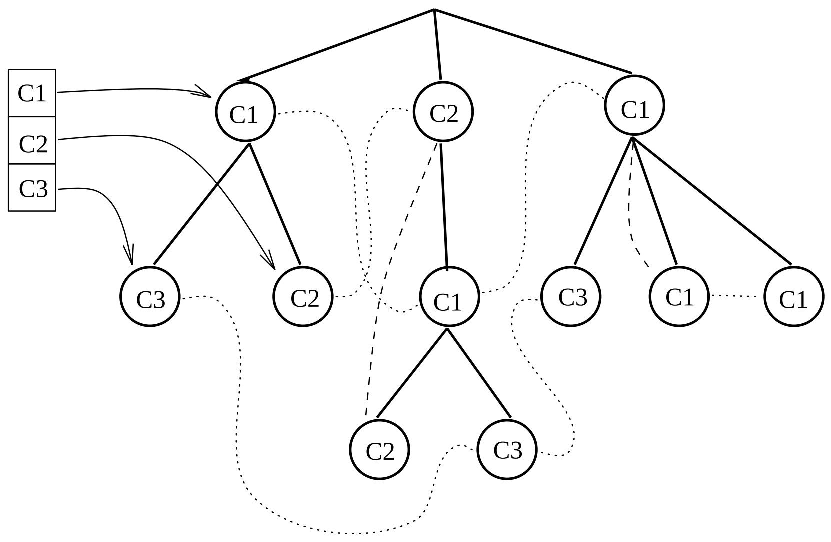
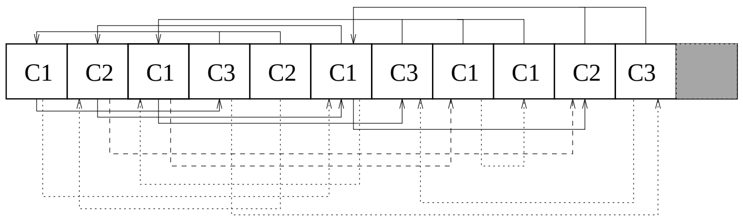


Figure 5.1: An example of a view tree. Full lines are the whole tree. Dashed lines are the per-constructor parent-child relationship. Dotted lines are the per-constructor sibling relationship. Arrows go from the array of constructors to the first top-level node of that constructor.

Memory Representation

A space-efficient way to store this view tree, both in memory and disk, is to store it in a breadth-first fashion on a long array, that is, all nodes from level i are before all nodes of level $i+1$. Also, we need an array of constructors, where for each element, among other data, we have the position in the view tree of the root of the first tree of nodes of that constructor (see Figure 5.2). This, combined with the mentioned child and sibling information, allows us to answer the query **Constr**(c) in time $O(|c|)$ (where $|c|$ stands for the number of nodes of the c constructor).

Note that the position of the parent of each node and of its first child is also stored in the array. The number of children needs not to be stored if the layout is breadth-first, since it is $first\ child(i+1) - first\ child(i)$; in order for this formula to work, the first child of a leaf node must be the same as the first child of the next



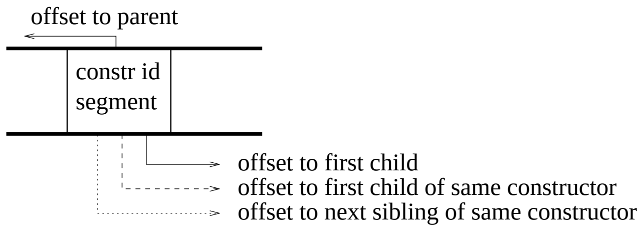

offset to parent
constr id
segment
offset to first child
offset to first child of same constructor
offset to next sibling of same constructor

Figure 5.2: The same view of Figure 5.1 in array form. The tree is laid in a breadth-first fashion. Below, the schema of a node with its 7 words of data.

non-leaf node, and we add a dummy final element whose first child is supposed to be itself. The array can be transformed into a tree (i.e. the format of answers to queries, as we see later) by a simple recursive algorithm that traverses the array following the child information.

Keeping the Index on Disk

The space requirement of this index is quite large, 7 words per node, so it is unlikely that it will fit in memory for real applications. If the array is kept on disk, we can obtain the tree of the view, but this time it is preferable to use a breadth-first traversal, to make a single pass over the array on disk. It is also possible to extract the tree of a constructor (query **Constr**(c)) by passing over (at worst all) the array and extracting in breadth-first the required tree (using a queue). Athough we pass over almost all the array, we read only sectors containing a node of the tree. Note that the sibling chain needs not to be in ascending order on the array (see C_2 in Figures 5.1 and 5.2), so it is not possible to predict the amount of seeks necessary to retrieve the subtree of a constructor.

In order to avoid this random access to the array, we need to store, for each node, the position of the next node (in the array) of the same constructor. With this information we can, in one pass over the array, read all nodes, store them in a hash table (accessing them by their position in the view array), and traverse the view as before, but searching the nodes in the hash table instead of randomly accessing the disk. The only problem is that we do not want to add more data to the index.

What we can do is to replace the two words that link the nodes of the same constructor by other two: the next element (in the array) of the same constructor, and the first ancestor of the same constructor. The idea is to retrieve the nodes of a given constructor by following the next-element link (thus making a single pass on the disk). We build the tree at the same time, and keep a hash table with the nodes, indexed by their position in the array. Each time a new element is needed from the disk, its first-ancestor (i.e. its parent in the tree of the constructor) is searched in the hash table, to determine from which node must it descend in the answer. This way we build the tree in expected linear time.

A word can be eliminated from this index by noting that, in fact, the information of where the first child of a node is can be eliminated at low cost. The only situation in which we need it is to answer the **View** query, but if the traversal is breadth-first, we can do it by using only the information on parent, with the same hash table technique.

Reindexing

Another important problem of this schema is related to reindexing. The ideal is to be able to reindex without having to rewrite the whole index, but only the modified part. As it is, the index must be almost wholly rewritten even if we add text at the end (which results in adding a subtree somewhere at the extreme right path of the view tree), or if we delete something at the end. We can achieve the ideal of rewriting only the modified part, but at the cost of incrementing seek time at query time. Since we are more interested on querying than on indexing efficiency, we prefer to keep the index more or less consecutive, and to search for a reasonable tradeoff.

This tradeoff can be achieved by modifying the layout of the index on disk to have a different (contiguous) file per level of the tree. This way, adding a subtree at the end can be done by appending the necessary nodes at the file of each level, and similarly to remove a subtree at the end. For adding or removing a subtree at the middle, the amount of data we must move corresponds to the number of nodes at the right of the path going from the root to the leaf of the modified zone, while with the initial layout we must move almost all the array (see Figure 5.3). We need to modify some parent/child pointers also, namely those of the parents of the moved nodes, which are the same nodes to move plus an additional path of the tree. While we move the nodes, we must update their parent/child information, and update the child information of the additional path.

Note that pointers can point now to another level, and thus we use absolute positions. However, a single word should still be enough to codify the level and position information.

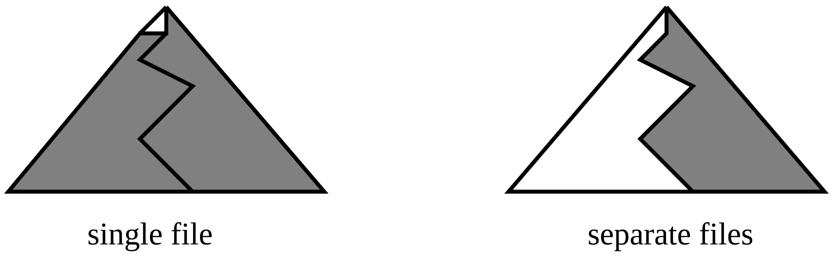


Figure 5.3: Reindexing overhead. The triangle represents the whole view, and the shaded part is what needs to be rewritten when a modification at the internal path occurs, under a single file and separate files policy, respectively.

Finally, we do not consider the work needed to obtain the new tree from the modified text, this is outside our interest here.

Avoiding Random Access to the Disk

Another problem is that some query operations need to access this index in unpredictable patterns, so the bigger the index, the more I/O needed to access arbitrary positions (be it due to paging or to accessing the disk at random). The reason for this access pattern is that the sets of nodes manipulated by the query algorithms are just pointers to the index, to minimize space utilization, and sometimes it is necessary to ask whether a node is the parent of another.

However, the parser of the query can sintactically determine which sets will need information on parentship, and for the sets that will need this information, read it to main memory at the time the sets are built (any set construction which is not from the text view starts with a **Constr** query). This information consists of the parent of each node, and is to be gathered for all the nodes involved in the right hand side of a **parent** operation, and in the left hand side of a **child** operation. This way, when those questions are made later, the information is already in main memory.

Favoring "Constr" Queries Against "View" Queries

Constr queries are much more common than **View** queries, while our index is more efficient for **View** than for **Constr**.

This problem can be overcome by having a separate index for each constructor, which is much like the index for the whole view we have been considering, but only the nodes of that constructor are stored. The difference is that we do not need to store explicitly where the sibling of the same constructor is, since that information is now implicit. We need instead to store where the first child and next sibling *in the view* are, and that may point to another file. So the pointers are now of the form *constructor:position*, what should fit in a single word. The *parent* pointer is as always, but in the *constructor:position* format.

Solving **Constr** with this schema is as easy as it is to solve **View** with the other: we make a single pass over the file of the constructor and build the tree, with no additional seek. To solve **View**, we must process the files of all constructors in parallel, advancing sequentially in each one. By having a pointer to the first top-level node of the view, we follow its pointers to its first child and next siblings, in the corresponsing file. If we traverse the view in DFS, then we will progress sequentially in each file. The change from file to file produces the only seeks of the process, which are not few. In this case, since the view must anyway fit in main memory, it is better to read all files to main memory and merge them there, where there are no seek costs.

This approach is better when **Constr** is more frequent than **View**, which is normally the case.

These files can also be split at each level to improve reindexing efficiency.

An interesting idea is that, since the query is known in advance, all the constructor trees could be read in a single pass over the disk, thus reducing the amount of disk I/O. This improvement can only be done if all leaves can be kept in memory at the same time and we are using full evaluation.

Lazy Evaluation

The single-file index is also suitable for lazy evaluation: if we need to expand only the first level of a given node from the tree of a view, all we need is to store for this node the pointer to the view array where its first child lies, and the number of children. This way, by using sequential disk access, we retrieve all its children from disk. If, instead, we need to expand a given node from the tree of a constructor, we store the pointer to the first child of the same constructor of the node in the index array, and from then on follow its siblings. If the constructor is spread along the view, this may take $O(d)$ disk accesses, where d is the number of children. Observe that the solution we proposed for this problem does not work for lazy evaluation, since the whole file has to be processed. The idea to eliminate a word from the index does not work either.

If we use the multiple-file index, then it becomes easy to expand the node of a **Constr** query, but hard to expand the node of a **View** query. In this case, expanding a **View** node involves accessing multiple files at random positions (and the solution of reading all into main memory does not work).

The proposed techniques to handle reindexing can be used here too.

5.1.2 A Partial Index

Under certain conditions, it is possible to use a much simpler index, which takes much less space and is equally efficient. Its problems are twofold: we cannot efficiently answer parent/child questions (i.e. they are $O(1)$ with the full index and $O(|N_V|)$ here), and we cannot allow two nodes to have the same segment, since we will not be able to distinguish which is ancestor of which. The idea is to store, for each constructor, two arrays holding one of them the initial positions of all segments of this constructor, and the other the final positions. Both of them are sorted by position (see Figure 5.4). Similar ideas are used in both the PAT text searching system and in SearchCity [ST92, Ars92].

The space utilization of this index is 2 words per node, so the whole space utilization is $2|N_V|$, much less

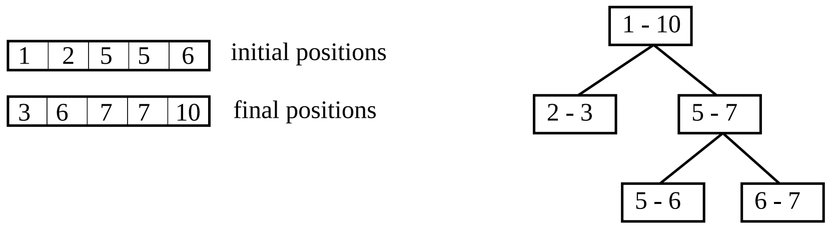


Figure 5.4: An example of a partial index and its associated view tree.

than the full index. To answer the query **Constr**(c), we select the file of c and traverse both arrays at the same time, in the same order as if we were merging the numbers of both files (in case of equality, we give preference to the final point). The output of this merge is manipulated as follows: we build the tree while reading the nodes, each time we get an initial position, we create a child of the current point, whose initial position is what we read; each time we get a final position, we complete the segment we have last opened, and we return to the previous level. This way, the tree is extracted from the two arrays in $O(|c|)$ time, and by sequentially accessing the files.

If, instead, we have to create the tree of the whole view, we need to interpret **View**(V) as **Constr**$(C_1) +$ ...+**Constr**(C_k), where $C_V = \{C_1, ..., C_k\}$ are the constructors of V. We show when implementing the $+$ operation that the cost to build $Q_1 + Q_2$ is $O(|Q_1| + |Q_2|)$, so supposing that the size of each constructor is $C = |N_V|/k$ (which is the worst case if we sum up from smaller to larger constructs), to sum all the k constructors until obtaining the whole view takes us

$$(C + C) + (2C + C) + ... + ((k-1)C + C) = C\left(\frac{k(k+1)}{2} - 1\right) \approx \frac{|C_V||N_V|}{2}$$

which is reasonable if we consider that queries on the whole view are not frequent.

Reindexing on this schema is quite easy: we take the new set of initial and final extremes and merge them with the current set (additions), or we extract from the current set the removed nodes (removals). In both cases, we must rewrite almost the whole tree.

A possible alternative to partial indexing is to use a PAT array to comprise both indices. This only works under the assumption that we have the text with markup, and that the initial and final mark of each constructor have a unique prefix (for example, the SGML-like `<constr>` and `</constr>`). In order to get the tree of a constructor, we search for `<constr>` and for `</constr>`, sort both results (since in the PAT array the ordering is lexicographic and not positional), and then we are in the same position as if we had our partial index. Thus, if the number of index points is n, this method takes a factor $O(2\log n + 2|c|\log|c|)$, on top of ours. Moreover, since the PAT array stores one word per entry and we need two entries per segment, the space utilization is the same as our partial index, at no benefit. Indeed, not having to index the markup reduces (slightly) the search time for normal matches. So this idea should be discarded.

Lazy Evaluation

This index performs well for full evaluation. However, if we want to expand just one node to get its children, it is very inefficient. To retrieve the first level of descendants we need to traverse the whole subtree of a constructor. For lazy evaluation we can use an index which is $O(3|N_V|)$ space. It is an adaptation of the full index that stores each constructor in a separate file, but storing only the segment and the first child of each node. This index offers the same efficiency that the previous one does, but now it is easy to expand just one node, in a similar way to the expansion in the full index, that is, with minimal disk movement and transfer. Expanding the node of a view is harder, as said previously.

5.1.3 Analysis of Indexing

In this section we analyze the space requirement for indices, and the time they need to answer **Constr** and **View** queries.

The initial proposal for the full index in memory uses 7 words. We show now how many bits it requires. From those 7 words, we have a segment description, which takes $2\log_2 T$ bits (T is the size of the text); a constructor id ($\log_2 |C_V|$ bits); and four pointers into the same index ($4\log_2 |N_V|$ bits). We also store an array of constructors with pointers to their first top-level nodes ($\log_2 |N_V|$ bits). So, the number of bits required for the full index is

$$|N_V|(4\log_2 |N_V| + 2\log_2 T + \log_2 |C_V|) + |C_V|\log_2 |N_V|$$

This index is optimal in time, since it answers $\mathbf{View}(V)$ in time $O(|N_V|)$ and $\mathbf{Constr}(c)$ in time $O(|c|)$. However, the assumption of keeping it in memory is not realistic. When keeping it on disk, the seek time can make the retrieval of $\mathbf{Constr}(c)$ to be $O(|c||N_V|)$ seek time.

If we add the information on the next (in the array) node of the same constructor, the seek time is reduced to $O(|N_V|)$, but we add $|N_V|\log_2 |N_V|$ to the space requirement (i.e. 8 words per node). We can avoid this by replacing the two fields as explained, thus keeping the original space requirement and average time complexity, although the worst case for **Constr** is now $O(|c|\log |c|)$ (by using appropriate data structures and external addressing in hashing).

Finally, we can eliminate the pointer to the first child in the view (thus eliminating another $|N_V|\log_2 |N_V|$ term in the space usage, i.e. using 6 words per node), and keep the same average complexity, but the worst case for $\mathbf{View}(V)$ turns to $O(|N_V|\log |N_V|)$.

The variations that store the index in different files do not alter the space usage, they just change the order of elements. However, the time complecity changes if we use a separate file for each constructor. In this case, we answer $\mathbf{Constr}(c)$ in time $O(|c|)$ in total, but $\mathbf{View}(V)$ can work $O(|N_V|^2)$ seek, although only $O(|N_V|)$ reads are made. In this case, for lazy evaluation it is better to translate $V = C_1 + ... + C_k$, and work $O(|N_V||C_V|)$, while for full evaluation it is better to read all files to main memory and merge there, where there are no seek costs.

Of course if we can pay the space cost of two indices, we can keep the best version for each query.

The partial index, instead, uses two words per node. In bits, it has two arrays of points, i.e. $2\log_2 T$ bits. We also need, for each constructor, to find its two arrays, so if we had all the data in a long tape, we would need to begin with $|C_V|$ "pointers" to the rest of the tape, i.e. $\log_2 |N_V|$ bits (the total number of positions of the rest of the tape). Thus, this index takes

$$2|N_V|\log_2 T + |C_V|\log_2 |N_V|$$

bits.

This index answers $\mathbf{Constr}(c)$ in $O(|c|)$ time, and $\mathbf{View}(V)$ in $O(|N_V||C_V|)$ time.

The lazy version of the partial index takes 3 words. In bits, it holds the segment and the next child, thus it takes

$$|N_V|(2\log_2 T + \log_2 |N_V|) + |C_V|\log_2 |N_V|$$

bits. The time complexity is the same as its 2-words version.

To get an idea of how big N_V can be with respect to T (the size of the database), the indexing structure of a 300-page book (with $T \approx 1\ Mb$) may have $N_V = 1000$ nodes (needing 28 Kb for the full index and 8 Kb for the partial one) if we index chapters, sections and paragraphs. If, instead, we index 1 Mb of C code with all the fine parsing details, we would have near 300K nodes, i.e. 8 Mb of space for the full index and 2.3 Mb for the partial one.

5.2 Querying

We describe here the algorithms to implement the defined operations, to show that all of them can perform very efficiently. Observe that we have already addressed the implementation of **View** and **Constr** as part of the definition of indices, so we focus on the rest here.

5.2.1 Full Evaluation

Since the result of a query is a set of nodes (a subset of some view), and the query language is compositional, all operations deal with sets of nodes to produce new sets. The generation of leaves is already addressed: **Constr** and **View** are resolved in the previous section, and since the algorithms for matching operations are not of interest in here, we just assume they generate a set of matches.

5.2.1.1 Data Structures

For efficient operation, we represent the set of nodes by a tree (it is already mentioned in the previous section, but with no justification for this). This tree does not represent any hierarchy, it is just a way to store a set, which allows efficient implementation of the operations.

The criterion to form the tree is straightforward: a node descends from another in the tree if and only if it does in the view (see Figure 5.5).

In order to save space, we could put the tree in an array, as we do with indices, but since these trees are going to be operated upon (i.e. additing, moving and deleting of nodes), we need a highly dynamic structure, which is not provided by the array implementation. Thus, we implement trees with pointers. For the same reason, it is difficult to have these trees on disk, because we need rapid access to different portions of them. Since the sets may be large, the need to store the full tree in memory is a weak point of this approach.

This problem can be solved by a virtual-memory-like approach, keeping part of the intermediate results (i.e. whole trees) 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 constructors. This idea, together with a good swapping policy, provides a uniform and elegant solution to the problem. Observe also that, while an operand has to be kept in memory to efficiently modify it, a read-only operand could be operated from disk with reasonable efficiency. This way, the policy could prefer the operands that are to be used just for reading, to swap them out.

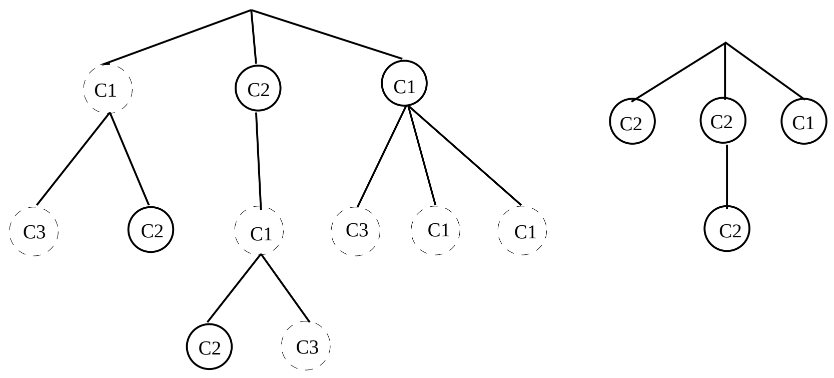


Figure 5.5: An example of a query (subset of the example view) and its tree representation.

From the nodes of the tree, we point to the information stored in the view array (see the section of indexing for the many variations on this view array).

Observe that, given the criterion to form the tree, any tree obtained from matches or operations on matches is flat, i.e. a linear array of non-overlapped, non-nested segments, so all operations on matches need to work on those arrays, while operations on other structures work over general trees.

5.2.1.2 Generating a Query Plan

Given the syntax tree of a query, we need to generate a *query plan*, that is, a sequence of operations to yield the desired result. There are many ways to linearize the operations of a tree preserving dependencies. We prefer the one which minimizes the total space needed for the computation. Suppose one has to evaluate X *op* $(Y$ *op* $(Z$ *op* $W))$. One could first obtain X and then obtain the other side. To obtain the other side one must, while keeping X in memory, compute Y, and then evaluate the other side, and so on. This way, we have, at a given time, X,Y,Z and W all in memory (of course we can paginate, but this is not desirable either). If, instead, we evaluate right-to-left in this case, we need space just for two operands at each time. Note that almost all operators select nodes from the left operand which have some property, so normally $|X \textit{ op } Y| \leq |X|$.

This problem, although simplified, is solved in [ASU86]: one has to obtain first the "heavier" operand, then the other, and then operate them. Since in [ASU86] the problem is assignment of registers to compile an expression tree, the "weight" of a tree is defined as its number of nodes. We need instead to estimate the size of our sets. In absence of good estimators, using the number of nodes seems a reasonable initial choice (although more sophisticated heuristics should be developed). Thus the algorithm would be as simple as solving the tree by selecting first the bigger subtrees to evaluate.

A useful modification to having a syntax tree for the query is to have a directed acyclic graph (DAG), to avoid re-evaluating common subexpressions. These are not so uncommon in our language: not only constructor names can be frequently repeated, but for example the *not* form is expressed as $Q - (Q \textit{ such that} \ldots)$.

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 [ASU86], which is known to be NP-Complete [GJ79].

Another important point is that we can write our algorithms to operate by modifying one of the operands to produce the answer, or by generating a new set. If the selected operand is to be used only once, it is better to modify it, otherwise we should generate a new set. This way, we should have two versions for each algorithm. Another alternative is to use modifying algorithms and make a copy of the selected operand before the operation; or to use generating algorithms and deleting the operand the last time it is used.

The query plan generator must implement the appropriate policy to avoid keeping unnecessary copies in memory, deleting operands the last time they are used. This can be avoided if the last use is "modifying", so a good policy is to leave that operation for the end. If we have both versions of the algorithms we just select the modifying version the last time we use an operand. Most of the algorithms modify just one operand, so the other is used just for reading. If we have only modifying operators, we can avoid making a copy if it is the last use for the operand to be modified. All these ideas must coexist with a general heuristic to evaluate a DAG minimizing the use of memory. This problem reveals of most importance in the experimental results, and deserves a complete study in the context of smart query plan generation.

A reasonable heuristic for a DAG [ASU86] is to collect the root and all shared nodes, forming the tree of each collected node, consisting of their descendant nodes, but stopping at leaves or at shared nodes (i.e. roots of other trees). Each tree is solved with the optimal algorithm for trees, and the order for evaluating trees is some topological ordering of the shared nodes, where after evaluating each node we must keep it until all its parents have been evaluated.

Observe that all these algorithms to select a suitable order of evaluation are meaningful only if we use full evaluation, since for lazy evaluation the order is dictated by the mechanism itself.

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 (see, for example, [CM94]).

5.2.1.3 Description of the Algorithms

Now we describe the algorithms to implement the operations. We define first the type of trees T_V, over view V, that these operators work on and yield. Operators are of type *operator*:

$$\begin{aligned} T_V &= Subtree_V^* \\ Subtree_V &= N_V \times T_V \\ operator &= T_V^* \to T_V \end{aligned}$$

As we mentioned before, we have two strategies for full evaluation.

The first one, called *search-operate*, operates by selecting one of the two operands and, for each of its nodes, searching into the (tree of the) other against what should it operate, making then the operation. The searcher/searched selection can be changed for the subproblems (subtrees), which are generally easy to find given the search performed by the root node. Which operand is searched into which is a heuristic decision. In the operands, we call "12" the strategy that searches the left operand into the right one, and "21" the opposite. For example, if we want to intersect P and Q (i.e. **is**), we can take each element p from the top-level of P, search it in the tree of Q, and keep it only if it is found in Q. The search for the children of p starts where the search for p ended. It seems a better idea to search the smaller tree into the bigger, so if the subtree where the children of p must be searched is smaller than the subtree of p we prefer to search the smaller Q subtree into the subtree of p. It also seems a good idea to have a data structure that allows binary search on each level of the trees. We have selected the heuristic to decide between "12" and "21" based on the results of the analysis, intuition and experimental results.

The second one, called *merge*, operates by sequentially traversing both trees, in a more or less synchronized way. For example, to solve (P **is** Q), we traverse both top-levels in synchronization, keeping the P nodes that we also find in Q. Depending on the inclusion relationships between P and Q nodes, we may have to descend in some of the two trees. This strategy works better with a data structure that allows efficient sequential access at each level.

As we show below, the worst-case behavior of *merge* is better than that of *search-operate*, but the average case has to be compared experimentally, since *search-operate* could be better if the operands have too different size. Along with the description of each algorithm we include its worst-case analysis.

We use the following notation (see also the formal definition of the model in Chapter 3):

- A segment can be written by denoting its two extremes $\langle a, b\rangle$, being a and b two natural numbers from 1 to T.
- $p > q$ means that the segment of p is definitely after the one of q, i.e. $From(p) > To(q)$. We use the same notation with p and q being segments, or even subtrees (denoting their root node). Analogously we define $p < q$.
- $p = q$ means that both *nodes* are the same (not segments) if they are from the same view, but if we compare two nodes from different views for equality, we mean segment equality.
- As expected, $\subseteq$ means $\subset$ or $=$ (recall the definition of $\subset$ between nodes).
- p *overlaps* q means $From(p) < From(q) \leq To(p) < To(q)$ or vice versa.
- We assume for convenience (in all *merge* and *lazy* algorithms) that each list of subtrees ends with a special segment $\langle\infty, \infty\rangle$, and we assume $\infty = \infty$.
- If P is a query, then its top-level list of nodes is referred as $\{p_1..p_{last}\}$. If the list of subtrees of P has only one element, this lone subtree is referred to as p. $\{p_a..p_b\}$ represents a subsequence of P.

- $node(t)$ and $query(t)$ are the node and subquery, respectively, of a $Subtree_V$ t. (this $query(t)$ may be not expanded in *lazy* algorithms).
- $|P|$ denotes the number of top-level elements of a tree, and $size(P)$ denotes the total number of nodes of P.
- If we compare two segments or a node against a segment, the comparison operator should be interpreted as if they were from different views. This is always well defined. For example, $\langle a,b\rangle \subset n \Leftrightarrow \langle a,b\rangle \subseteq Segm(n)$.
- $parent_{view}(node)$ denotes the parent of $node$ in the view tree, and $parent_Q(node)$ denotes the same in the tree of the query Q; it can be $none$ if the node has no parent. $parent_Q$ is not stored but computed while searching the $node$ into Q.
- pos_{view} and pos_Q denote the position of the child into its corresponding $parent$.
- $none$, used as a query, denotes an empty query.
- When we use an operator name in prefix form, we indicate that the operation is not performed but left specified (for lazy evaluation).
- If $P = \{p_1..p_n\}$ and $Q = \{q_1..q_m\}$ are queries (denoted by their top-level trees), then the query $P : Q$ (concatenation of top-levels) denotes $\{p_1..p_n, q_1..q_m\}$. This can only be done if $p_n < q_1$.
- We use a C-like notation for our algorithms, with call-by-value convention and replacing braces by indentation. If any statement follows an `If` or `While` condition in the same line, then all the statements following the condition in the same line form the body of the compound sentence. In *merge* and *lazy* versions, we use a `switch`-like notation (a big left brace), in which the conditions are sequentially tested.

Finally, recall that we are not deciding whether the operations work by modifying its arguments or they generate a new set with the result (we use the modifying or the generating terminology, which is most convenient for the exposition), and that we are not including in the algorithms the operations needed to keep count of sizes (which we need for some operations). We disregard also the consideration of some exception conditions that can happen, to avoid complicating the exposition (e.g. empty queries, empty segments, null values, unmarking, etc.).

If we use modifying operators, we need to merge nodes from different levels, extract a range of nodes from a level, etc. Although we do not detail this, we assume we use a data structure that allows searching, extracting a subinterval and merging (when the merged set is to be inserted entirely in a single point) in logarithmic time. For example, we can use balanced binary trees. Searching can be done by simple binary search. We can delete a subinterval by making two "cut" operations and then merging the resulting subtrees (i.e. to delete $(a..b)$ from T, we make $(L_a, R_a) = Cut(T, a)$, then $(I_{ab}, R_b) = Cut(R_a, b)$, and then $T' = Merge(L_a, R_b)$). Finally, we can insert a subinterval by making one "cut" and merging three subtrees, i.e. to insert I_{ab} in T, we make $(L, R) = Cut(T, a)$ (note that since the whole subinterval fits in the same point of T, $Cut(T, a) = Cut(T, b)$), then we make $T' = Merge(L, Merge(I_{ab}, R))$. All this can be done in $O(\log n)$ time for trees of size n, including the necessary rebalancing.

We use the following numbers in the analysis: n_X is the size of the set corresponding to operand X, h_X is the height of its tree (in the worst case it can be n_X) and d_X is the maximum degree of its tree (it can also be n_X in a flat tree). We also use n, d and h as the maximun corresponding value between all operands (there are two or three operands). Although it is possible to obtain a measure in terms of n only, there exists normally some relationship to h and d, which should be interesting when we know some properties of the trees a particular application uses.

Some heuristic assumptions we have to make when we attempt to map the results of the analysis into a heuristic decision for "12" vs "21" (since we do not store d nor h) are: $d, h, h \log d \approx \log n$, $\log d \approx constant$.

"+" operator

Search-operate version: We select one of the two trees to "put into" the other. For each element of the top-level of one tree, we search it into the other, inserting it where corresponds, and chaining with subproblems appropriately. Which tree is searched into which is a heuristic decision, as in all *search-operate* algorithms.

To partition the problem adequately, we do not search each element of the first tree in sequence, but we partition the set in a binary fashion. This way, if we have to insert $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$ into $\{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$, we operate by first searching $\langle 1,n\rangle$, then $\langle 1,n/2\rangle$ and $\langle n/2+1,n\rangle$, then $\langle 1,n/4\rangle$, $\langle n/4+1,n/2\rangle$, $\langle n/2+1,3n/4\rangle$ and $\langle 3n/4+1,n\rangle$, etc., until reaching the real nodes. Note that in this way we insert the node $\langle 1,n\rangle$ until the bottom of the second operand, making n operations in total, while if we inserted each element $\langle i,i\rangle$, we would make n^2 operations. One can see this technique as if for each level of the searching tree, we built a binary search tree, adding artificial internal nodes (e.g. $\langle 1,n\rangle$, $\langle n/2+1,3n/4\rangle$). Note that we add $O(n)$ artificial nodes. This technique is applied in all the *search-operate* algorithms.

See Figure 5.6 for another example. From a single level of P a binary tree of artificial nodes is created. When inserting that level into Q, an originally $O(n^2)$ task (straight arrows) is converted to an $O(n \log n)$ task (curve arrows).

Observe that no overlaps are possible, since P and Q must be from the same view.

$P+Q$ (*search-operate* version): Operate (P,Q)

```
Operate $(P,Q)$

If $(size(P) < size(Q))$ return Plus $(P,Q)$
else Plus $(Q,P)$

Plus $(P,Q)$

If $|P| = 1$
   $(Q',b,t) \leftarrow$ Search $(node(p),Q)$.
   If $(q'_b \neq p)$ $Q' \leftarrow \{q'_1..q'_{b-1}\} : (p, \{q'_b..q'_t\}) : \{q'_{t+1}...\}$.
   $query(q'_b) \leftarrow$ Operate $(query(q'_b), query(p))$.
else
   $(Q',b,t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, Q)$.
   $Q' \leftarrow \{q'_1..q'_{b-1}\}$ : Plus $(\{p_1..p_{half}\}$, Plus $(\{p_{half+1}..p_{last}\}, \{q'_b..q'_t\}))$ : $\{q'_{t+1}...\}$.
Return $Q'$.
```

```
Search $(node,Q)$

$b \leftarrow 1$.   $t \leftarrow |Q|$.
Repeat
   Restrict $b,t$ while $q_{b-1} < node < q_{t+1}$ (binary search).
   If $(view(node) = view(q_b) \neq text\ \ view)$
      If $(node \not\subset q_b)$ Return $(Q,b,t)$.
   else If $(Segm(node) \not\subset Segm(q_b))$ Return $(Q,b,t)$.
   $Q \leftarrow query(q_b)$.
```

```
Searchi ($node, Q$)

$b \leftarrow 1$.   $t \leftarrow |Q|$.
Repeat
   Restrict $b, t$ while $q_{b-1} < node < q_{t+1}$ (binary search).
   If ($node \not\subseteq q_b$) Return ($Q, b, t$).
   $Q \leftarrow query(q_b)$.
```

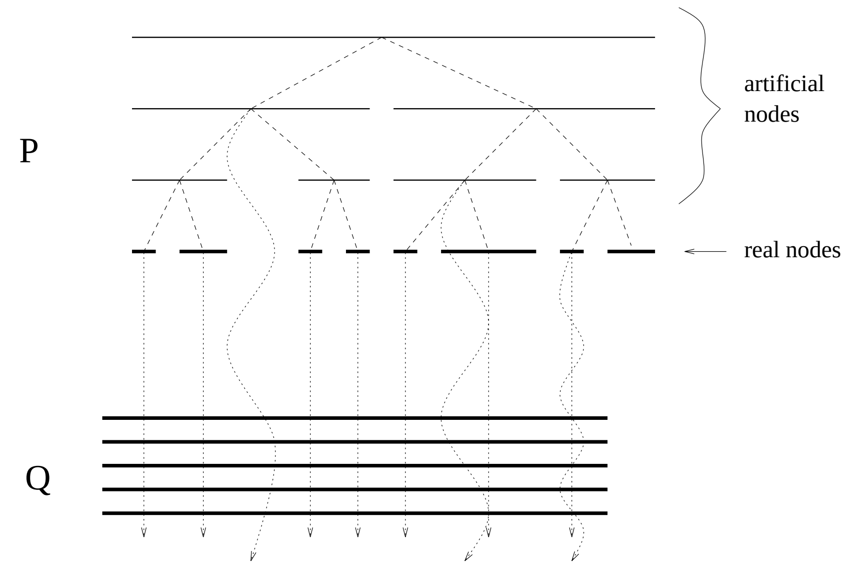


Figure 5.6: An example of the mechanism of artificial nodes.

`Search` operates by binary searching the left and right extremes of the node, thus it is $O(\log d)$. Note that if `Search` takes care of, prior to searching each side, verifying if the answer is not already the initial value of b or t, no additional work is done by adding the artificial nodes, since at each iteration, two of the four ends to search are already computed: after determining that $\langle From(p_1), To(p_{last})\rangle$ restricts the Q list to $\{q_b..q_t\}$, we search $\langle From(p_1), To(p_{half})\rangle$ in $\{q_b..q_t\}$ (then $From(p_1)$ is work already done) and $\langle From(p_{half+1}), To(p_{last})\rangle$ in the same array (then $To(p_{last})$ is work already done). In this way, we do not make any additional work. `Searchi` is similar, but it enters into a node also when it is equal to the searching node.

We can show that this algorithm is between $O(n \log d)$ and $O(n \log^2 d)$. We have not been able to tighten this analysis. This analysis is only valid if both operands are from the same view, which is true in this case.

We first analyze the algorithm as if always P was searched into Q, we take care later of the possible reversing of roles in subproblems.

To see that it is $O(n \log^2 d)$ we first demonstrate that a given node of Q can only be traversed by $O(\log d_P)$ nodes from P. We are interested in nodes of P that traverse some node of Q (by traversing we mean that along the global execution of `Plus`, the P-node searches into a level of Q and falls inside one Q-node, going to the next level; this Q-node is said to be traversed by the P-node), since the P-nodes which do not traverse Q-nodes can only search on a single level of Q, thus their whole complexity is $O(n_P \log d_Q)$.

To see that, we first show that all the P-nodes traversing a single Q-node are in a single level of P. Suppose p_1, p_2 traverse q, and are from different levels of P. Suppose first that both nodes have parents, which we denote P_1, P_2. If p_i traverses q, then P_i does not (since if P_i traversed q, p_i would search into the descendants of q, being impossible for it to traverse q). Note also that P_i cannot be included in q, since p_i is the first on the path from p_i to the top level of P in traversing q. This also implies that p_1, p_2 cannot descend from one another, that is, they must be disjoint. Now observe that we have derived the following facts: p_1 and p_2 are disjoint and included in q, p_i is included in P_i, and P_1 and $P2$ are not included in q. This is impossible, since all these nodes belong to a single hierarchy (view), except if $P_1 = P_2$, what we wanted to demonstrate. The generalization to the case in which some p_i has no parent is trivial.

Once we have this result, we show that with the method of artificial nodes, only $O(\log d_P)$ (artificial or original) nodes can traverse q. We reason as if the level of P under consideration was a binary tree (of depth $\log d_P$), to show that at most 2 nodes of each level of P can traverse q.

Suppose there were 3 nodes of a single level of the binary tree traversing q. If they are not contiguous, then all nodes of the same level between the first and the last of the three must be included in q also. Thus, there are at least 3 contiguous nodes in a single level included in q. From those 3 nodes, two of them must have the same parent (an artificial node), which extends from the start of the first one to the end of the second one. Therefore, this parent node must also be included in q, moreover, properly included (because of the third node). Then, this parent node should traverse q, and not its two children. A contradiction.

Now, the rest is easy. Since any node of Q is traversed by at most $O(\log d_P)$ nodes, if we count the search to determine each traversal, we have that the total work is $O(n_Q \log d_P \log d_Q)$. Summing this to the work done by P nodes that do not traverse a Q level, we have a total complexity of $O(\log d_Q(n_P + n_Q \log d_P)) = O(n \log^2 d)$.

The easiest way to take switching into account (i.e. the fact that `Operate` can change the order between P and Q) is to consider the total number of calls of the form `Plus`(P,Q) and the total of the form `Plus`(Q,P). Both totals are less than if all the calls were made by only one form, thus at worst we double the time (of course this is not true, a better approximation is to say that summing both complexities we complete $n \log^2 d$, but for our purposes it is the same). Therefore, the complexity remains the same if we see it in terms of d and n.

To see that it is no less than $O(n \log n)$, consider $\{\langle 1, n-1\rangle, \langle n, n\rangle, \langle 1, n-2\rangle, \langle n-1, n-1\rangle, ..., \langle 1,1\rangle, \langle 2,2\rangle\} + \{\langle 1,1\rangle, \langle 2,2\rangle, ...\langle n,n\rangle\}$.

Merge version: We merge both trees, at the top level first. When one segment is included in another, we descend on the corresponding level to merge the included segments.

$P + Q$ (*merge* version): Operate (P, Q, ∞).result

```
Operate $(P, Q, limit)$

$res \leftarrow \lambda$.
While $\min(To(p_i), To(q_j)) < limit$

  $p_i < q_j$ :  $res \leftarrow res : p_i$.  Pass $i$.
  $p_i > q_j$ :  $res \leftarrow res : q_j$.  Pass $j$.
  $p_i = q_j$ :  $res \leftarrow res : (node(p_i)$, Operate($query(p_i), query(p_j), \infty$).$result$).  Pass $i, j$.
  $p_i \subset q_j$ :  $Op \leftarrow$ Operate ($\{p_i ...\}, query(q_j), To(q_j)+1$).
             $res \leftarrow res : (node(q_j), Op.result)$.  $\{p_i ...\} \leftarrow Op.rest$.  Pass $j$.
  $p_i \supset q_j$ :  Exchange $\{p_i ...\} \leftrightarrow \{q_j ...\}$.

Return ($result = res, rest =$ nonempty list from $\{p_i ...\}$ and $\{q_j ...\}$).
```

This algorithm is linear, which can be proven by a simple argument: at each comparison we make, at least one element from P or Q is solved and no longer compared. This way, we have $O(n_P + n_Q) = O(n)$ time. To see this, we can inductively assume that `Operate` is linear (which is true when no further recursive invocations are made, since for each operation at least one element is eliminated from the problem), and observe that it returns the part of the list on which it has not worked. This way, since the recursive invocations are linear and the elements they consider are not reconsidered by the caller, we have total linear behavior.

To see that the algorithm is no less than $O(n)$ consider $\{\langle 1,1\rangle, \langle 3,3\rangle, ..., \langle 2n+1, 2n+1\rangle\} + \{\langle 2,2\rangle, \langle 4,4\rangle, ..., \langle 2n+2, 2n+2\rangle\}$ (both flat trees).

"−" operator

Search-operate version: We select one tree to search it into the other. We delete elements from P if we find them in Q also.

Observe that no overlaps are possible, since P and Q must be from the same view.

$P - Q$ (*search-operate* version): Operate (P, Q)

```
Operate $(P, Q)$

If $(size(P) < size(Q))$ Minus12 $(P, Q)$.
else Minus21 $(P, Q)$.
```

```
Minus12 $(P, Q)$

If $|P| = 1$
    $(Q, b, t) \leftarrow$ Search $(node(p), Q)$.
    Operate $(query(p), \{q_b..q_t\})$.
    If $(q_b = p)$ $P \leftarrow query(p)$.
else
    $(Q, b, t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, Q)$.
    Operate $(\{p_1..p_{half}\}, \{q_b..q_t\})$.  Operate $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\})$.
```

```
Minus21 $(P, Q)$

If $|Q| = 1$
    $(P, b, t) \leftarrow$ Search $(node(q), P)$.
    Operate $(\{p_b..p_t\}, query(q))$.
    If $(q = p_b)$ $P \leftarrow \{p_1..p_{b-1}\} : query(p_b) : \{p_{t+1}...\}$.
else
    $(P, b, t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
    Operate $(\{p_b..p_t\}, \{q_1..q_{half}\})$.  Operate $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\})$.
```

The analysis of this algorithm is similar to the corresponding "+" operator, thus we have $O(n \log d..n \log^2 d)$.

Merge version: We traverse both trees in synchronization. When we find a match, we delete the corresponding node from P. The inclusions among segments drive our movement through levels.

The algorithm is presented in the next page.

This algorithm is linear, which can be proven by an argument very similar as for "+": at each comparison, at least one element is left out of the problem, `Operate` is linear and does not repeat the work done in its recursive calls. Therefore, we have $O(n_P + n_Q) = O(n)$ time.

To see that is no less than $O(n)$ consider the same example as for "+".

$P - Q$ (*merge* version): Operate (P, Q, ∞).result

```
Operate $(P, Q, plimit)$

$res \leftarrow \lambda$.
While $To(p_i) < plimit$

  $p_i < q_j$ :   $res \leftarrow res : p_i$.   Pass $i$.
  $p_i > q_j$ :   Pass $j$.
  $p_i = q_j$ :   $\{p_i ...\} \leftarrow query(p_i) : \{p_{i+1} ...\}$.
  $p_i \subset q_j$ :   $Op \leftarrow$ Operate $(\{p_i ...\}, query(q_j), To(q_j) + 1)$.
                $res \leftarrow res : Op.result$.   $\{p_i ...\} \leftarrow Op.restp$.   Pass $j$.
  $p_i \supset q_j$ :   $Op \leftarrow$ Operate $(query(p_i), \{q_j ...\}, \infty)$.
                $res \leftarrow res : (node(p_i), Op.result)$.
                $\{q_j ...\} \leftarrow Op.restq$.   Pass $i$.

Return $(result = res, restp = \{p_i ...\}, restq = \{q_j ...\})$.
```

"is" operator

Search-operate version: We select one tree to search it into the other. We mark elements of P when we find them in Q also, later we delete unmarked elements.

Observe that no overlaps are possible, since P and Q must be from the same view.

P **is** Q (*search-operate* version): Operate (P, Q)

```
Operate $(P, Q)$

MarkIs $(P, Q)$.
Delete unmarked elements from $P$.

MarkIs $(P, Q)$

If $(size(P) < size(Q))$ MarkIs12 $(P, Q)$.
else MarkIs21 $(P, Q)$.
```

```
MarkIs12 $(P, Q)$

If $|P| = 1$
   $(Q, b, t) \leftarrow$ Search $(node(p), Q)$.   If $(q_b = p)$ Mark $p$.
   MarkIs $(query(p), \{q_b .. q_t\})$.
else
   $(Q, b, t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last}) \rangle, Q)$.
   MarkIs $(\{p_1 .. p_{half}\}, \{q_b .. q_t\})$.   MarkIs $(\{p_{half+1} .. p_{last}\}, \{q_b .. q_t\})$.
```

```
MarkIs21 $(P, Q)$

If $|Q| = 1$
   $(P, b, t) \leftarrow$ Search $(node(q), P)$.   If $(q = p_b)$ Mark $p_b$.
   MarkIs $(\{p_b .. p_t\}, query(q))$.
else
   $(P, b, t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last}) \rangle, P)$.
   MarkIs $(\{p_b .. p_t\}, \{q_1 .. q_{half}\})$.   MarkIs $(\{p_b .. p_t\}, \{q_{half+1} .. q_{last}\})$.
```

The analysis of this algorithm is much as for the corresponding "+" operator, except that we have to add a $O(n_P)$ to the complexity, for collecting marked nodes. Since this does not change the total, we have $O(n \log d..n \log^2 d)$.

Merge version: We traverse both trees in synchronization. When we find a match, we include the corresponding node of P. The inclusions among segments drive our movement between levels.

P **is** Q (*merge* version): Operate (P, Q, ∞).result

```
Operate $(P, Q, plimit)$

$res \leftarrow \lambda$.
While $To(p_i) < plimit$
    $p_i < q_j$ :   Pass $i$.
    $p_i > q_j$ :   Pass $j$.
    $p_i = q_j$ :   $res \leftarrow res : (node(p_i),$Operate$(query(p_i), query(q_j), \infty).result)$.   Pass $i, j$.
    $p_i \subset q_j$ :   $Op \leftarrow$ Operate $(\{p_i...\}, query(q_j), To(q_j)+1)$.
                  $res \leftarrow res : Op.result$.   $\{p_i...\} \leftarrow Op.rest$.   Pass $j$.
    $else$ :        $\{p_i...\} \leftarrow query(p_i) : \{p_{i+1}...\}$.
Return $(result = res, rest = \{p_i...\})$.
```

This algorithm is $O(n_P + n_Q) = O(n)$, again with the argument that any comparison deletes at least one element from the problem.

To see that it is no less than $O(n)$, we can use the same example of the "+" operator.

"same" operator

This operation is exactly like **is**, except in that we do not check for node equality but for segment equality. In this case, overlaps are possible. This fact does not affect the algorithms (observe that the clause $p_i = q_j$ of the *merge* version is tested before $p_i \subset q_j$), but the analysis done for the *search-operate* version is no longer valid, because of overlaps.

In this case, we analyze `MarkIs12` as follows: each element of P can traverse a whole path of Q, making $\log d_Q$ operations on each level; but it can never traverse more than the whole Q. This way, we have $O(n_P \min(n_Q, h_Q \log d_Q))$. `MarkIs21` is similar, that is, $O(n_Q \min(n_P, h_P \log d_P))$. Thus, switching produces $O(n \min(n, h \log d))$.

We can show that it can reach $O(n^2)$ with this example: $\{\langle 1, n\rangle, \langle n+1, n+1\rangle, \langle 1, n-1\rangle, \langle n, n\rangle, ..., \langle 1, 1\rangle, \langle 2, 2\rangle\}$ **same** $\{\langle 2, 2n+1\rangle, \langle 2, 2n\rangle, ..., \langle 2, n+2\rangle\}$. All the nodes $\langle i, i\rangle$ traverse the whole chain of segments of Q; and this happens at least for $i \in n..n/2$ (because of switching).

However, observe that our analysis for the *merge* version does remain the same, that is, $O(n_P + n_Q) = O(n)$.

"in" operator

Search-operate version: We can search each element of P in the top-level of Q to determine whether it is included or not, in the last case we replace the P node by its children; or we can search each element of the top-level of Q into P to determine which nodes of P it includes, mark them, and later collect the marked P-nodes.

Observe that overlaps are possible here, since P and Q can be from different views.

P **in** Q (*search-operate* version): Operate (P, Q)

```
Operate $(P,Q)$

MarkIn $(P,Q)$.
Delete unmarked elements from $P$ (if a node is marked, its descendants are
considered marked).

MarkIn $(P,Q)$

If $(size(P)\log_2|Q| < |Q|\log_2(size(P)))$ MarkIn12 $(P,Q)$.
else MarkIn21 $(P,Q)$.
```

```
MarkIn12 $(P,Q)$

If $|P| = 1$
    Restrict $b,t$ while $q_{b-1} < p < q_{t+1}$.
    If $(p \subset q_b)$ Mark $p$.
    else MarkIn $(query(p), \{q_b..q_t\})$.
else
    Restrict $b,t$ while $q_{b-1} < \langle From(p_1), To(p_{last})\rangle < q_{t+1}$.
    MarkIn $(\{p_1..p_{half}\}, \{q_b..q_t\})$.  MarkIn $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\})$.
```

```
MarkIn21 $(P,Q)$

If $|Q| = 1$
    $(P,b,t) \leftarrow$ Searchi $(node(q), P)$.
    If $(q$ $overlaps$ $p_b)$ MarkIn21 $(query(p_b), \{q\})$.  $b \leftarrow b+1$.
    If $(q$ $overlaps$ $p_t)$ MarkIn21 $(query(p_t), \{q\})$.  $t \leftarrow t-1$.
    Mark $p_b..p_t$.
else
    $(P,b,t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
    MarkIn $(\{p_b..p_t\}, \{q_1..q_{half}\})$.  MarkIn $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\})$.
```

The analysis of this algorithm is as follows.

The final deletion of unmarked nodes is $O(n_P)$.

For MarkIn12, in the worst case we would have to process the whole P against the top-level of Q. This implies that it is $O(n_P \log d_Q)$.

To see that MarkIn12 can reach $O(n \log n)$ consider $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ...\langle 1, n+1\rangle\}$ **in** $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$. Observe that in this analysis, d_Q stands in fact for the arity of the top-level, not the maximum arity.

For MarkIn21, we can use the same analysis done for "+", since although we needed there to assume the absence of overlaps to show that only the children of a single node (of Q in this case) could traverse a node (of P in this case), and there may be overlaps here, we have only the top-level nodes in this case. Thus, it also holds here that fact. The rest of the proof did not need the absence of overlaps, so we have that only $\log d_Q$ nodes can traverse a given P node. If we add the nodes of d_Q that do not traverse a level in P, we have a total of $O(\log d_P(d_Q + n_P \log d_Q)) = O(n \log^2 d)$.

We have still not considered the marking done at each level by MarkIn21. Observe that the marking of the range $p_b..p_t$ is $O(d_P)$ in principle, but we can use a balanced binary tree, where the root stands for marking $p_1..p_d$, its children for $p_1..p_{d/2}$ and $p_{d/2+1}..p_d$, etc. This way, marking involves traversing that binary tree from the root to the leaves, a $O(\log d_P)$ task, thus this marking is $O(d_Q \log d_P)$ in total and does not affect the whole result.

Again, we have not been able to show that it can reach $O(n \log^2 n)$, we can only reach $O(n \log n)$ by considering the same example as for "+".

If faced to the problem of having best worst-case complexity, we use only `MarkIn12` to achieve $O(n \log d)$ behavior.

This analysis does not change if P and Q are from the same view.

Merge version: We traverse the top levels of P and Q, in synchronism. When a top-level node of P is included in one of Q, that subtree of P is included. When a top-level node of P includes one of Q, we replace the node of P by its children.

P **in** Q (*merge* version): Operate (P, Q)

```
Operate $(P,Q)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$
  $p_i < q_j$ :   Pass $i$.
  $p_i > q_j$ :   Pass $j$.
  $p_i \subset q_j$ :   $res \leftarrow res : p_i$.   Pass $i$.
  $else$ :        $\{p_i ...\} \leftarrow query(p_i) : \{p_{i+1} ...\}$.
Return $res$.
```

This algorithm is $O(n_P + n_Q) = O(n)$, which can be seen by observing that any element is considered at most once.

We can refine this analysis as follows: each time a node of P is expanded is because it contains an element of Q or because it overlaps with an element of Q, thus each extreme of each segment of (the top-level of) Q is compared, at most, with a complete path of P (length h_P). That is because once we descend the first level in P, the relevant list from the top-level of Q has only one element (the original q_j). At each level of this path, the merge can take us d_P comparisons, thus the cost is $O(d_Q h_P d_P) = O(d^2 h)$. That means, for example, that in a model with constant d and balanced trees the operation takes $O(\log n)$.

This way, the complexity of this operator is $O(\min(n_P + n_Q, d_Q h_P d_P)) = O(\min(n, d^2 h))$.

To see that it can reach $O(n)$, we use the same example of "+".

This analysis does not change if P and Q are from the same view.

"beginin/endin" operators

Search-operate version: The idea is quite the same as for **in**. We use here a second mark, which means that the node is marked but its children are not necessarily marked. Overlaps are also possible here.

```
MarkIn12 $(P,Q)$

If $|P| = 1$
   Restrict $b,t$ while $q_{b-1} < p < q_{t+1}$.
   If $(p \subseteq q_b)$ Mark $p$.
   else
      If $(From/To(p) \in q_b/q_t)$ Mark2 $p$.
      MarkIn $(query(p), \{q_b..q_t\})$.
else
   Restrict $b,t$ while $q_{b-1} < \langle From(p_1), To(p_{last}) \rangle < q_{t+1}$.
   MarkIn $(\{p_1..p_{half}\}, \{q_b..q_t\})$.   MarkIn $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\})$.
```

```
MarkIn21 $(P, Q)$

If $|Q| = 1$
   $(P, b, t) \leftarrow$ Search $(node(q), P)$.
   If $q$ overlaps $p_b$
      [ONLY beginin] Mark2 $p_b$.
      MarkIn21 $(query(p_b), \{q\})$.   $b \leftarrow b + 1$.
   If $q$ overlaps $p_t$
      [ONLY endin] Mark2 $p_t$.
      MarkIn21 $(query(p_t), \{q\})$.   $t \leftarrow t - 1$.
   Mark $p_b..p_t$.
else
   $(P, b, t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
   MarkIn $(\{p_b..p_t\}, \{q_1..q_{half}\})$.  MarkIn $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\})$.
```

Operate and MarkIn are almost the same, except when deleting unmarked nodes, when we keep the elements with the second mark, but we go into their subtrees, since they are not automatically marked.

The analysis for these algorithms is exactly the same as for **in**.

Merge version: The algorithms are quite the same as for **in**.

P **beginin** Q (*merge* version): Operate (P, Q).result

```
Operate $(P, Q)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

   $p_i < q_j$ :          Pass $i$.
   $p_i > q_j$ :          Pass $j$.
   $p_i \subset q_j$ :          $res \leftarrow res : p_i$.  Pass $i$.
   $From(p_i) \in q_j$ :  $Op \leftarrow$ Operate $(query(p_i), \{q_j...\})$.
                          $res \leftarrow res : (node(p_i), Op.result)$.  $\{q_j...\} \leftarrow Op.rest$.  Pass $i$.
   $else$ :               $\{p_i...\} \leftarrow query(p_i) : \{p_{i+1}...\}$.

Return $(result = res, rest = \{q_j...\})$.
```

P **endin** Q (*merge* version): Operate (P, Q)

```
Operate $(P, Q)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

   $p_i < q_j$ :  Pass $i$.
   $p_i > q_j$ :  Pass $j$.
   $p_i \subset q_j$ :  $res \leftarrow res : p_i$.  Pass $i$.
   $else$ :       $Op \leftarrow$ Operate $(query(p_i), \{q_j...\})$.
                  $\{q_j...\} \leftarrow Op.rest$.  While $(To(q_j) < To(p_i))$ Pass $j$.
                  If $(To(p_i) \in q_j)$ $res \leftarrow res : (node(p_i), Op.result)$.
                  else $res \leftarrow res : Op.result$.
                  Pass $i$.

Return $(result = res, rest = \{q_j...\})$.
```

The analysis of these algorithms is the same as for **in**. The only difference is that there is some extra work in retraversing Q (i.e. the clause $p_i > q_j$). But observe that it is, in total, linear in the size of the top-level of Q. Thus, we add $O(d_Q)$, which does not affect the total.

To see that it can reach $O(n)$, we use the same example as for "+".

If P and Q are from the same view, **beginin/endin** should be interpreted as **in**.

"[s] in/beginin/endin" operators

Search-operate: We can take each element of Q, search it into P, and mark the s-th immediate descendants.

Or we can take each element of P, search in Q its including nodes, and for each including node, we search back in P its immediate descendant, marking the s-th ones. This is done until there are no more including nodes of the P-node in Q, or until the P-node is marked (each element in P is given its opportunity to be marked). The used elements of Q are also marked, to avoid repeating work (we collect in an implicit list the marked Q elements, to avoid traversing the whole Q to unmark it). Due to the form of traversing Q, we cannot use for a node of P a node of Q corresponding to a higher level of P (since when a P-node gives the turn to its descendants, it has traversed in Q all the levels that corresponded to it).

Observe that overlaps are possible here, since P and Q need not to be from the same view.

[s] P **in** Q (*search-operate* version): Operate (P, Q)

```
Operate $(P,Q)$

MarkIn $(P,Q)$.
Delete unmarked elements from $P$, unmark elements of $Q$.

MarkIn $(P,Q)$

If $(size(P) < size(Q))$ MarkIn12 $(P,Q)$.
else MarkIn21 $(P,Q)$.
```

```
MarkIn12 $(P,Q)$

Repeat
   Restrict $b,t$ while $q_{b-1} < \langle From(p_1), To(p_{last})\rangle < q_{t+1}$.
   If $(|P| = 1)$ If $p \not\subset q_b$ Break repeat.
   else If $(\langle From(p_1), To(p_{last})\rangle \not\subset q_b)$ Break repeat.
   If $q_b$ is unmarked
      Mark $q_b$.
      Restrict $base, top$ while $p_{base-1} < q_b < p_{top+1}$.
      [EXCEPT endin] If $(q_b$ $overlaps$ $p_{base})$ $base \leftarrow base + 1$.
      [EXCEPT beginin] If $(q_b$ $overlaps$ $p_{top})$ $top \leftarrow top - 1$.
      For each $m \in base..top$
         If $(p_m$ is unmarked $\wedge$ $m - base \in s)$ Mark $p_m$.
   $Q \leftarrow query(q_b)$.
If $(|P| = 1)$ MarkIn $(query(p), \{q_b..q_t\})$.
else MarkIn $(\{p_1..p_{half}\}, \{q_b..q_t\})$.  MarkIn $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\})$.
```

```
MarkIn21 $(P,Q)$

If $|Q| = 1$
   If $q$ is unmarked
      $(P,b,t) \leftarrow$ Search $(node(q), P)$.
      Mark $q$.  MarkIn $(\{p_b..p_t\}, query(q))$.
      [EXCEPT endin] If ($q_b$ $overlaps$ $p_b$) $b \leftarrow b+1$.
      [EXCEPT beginin] If ($q_b$ $overlaps$ $p_t$) $t \leftarrow t-1$.
      For each $m \in b..t$
         If $(m - b \in s)$ Mark $p_m$.
else
   $(P,b,t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
   MarkIn $(\{p_b..p_t\}, \{q_1..q_{half}\})$.  MarkIn $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\})$.
```

We analyze this algorithm by considering both alternatives separately. Note first that the final collection of marked nodes is $O(n_P)$ (although we have to unmark nodes of Q also, observe that the marked nodes of Q are kept in a list to unmark them, and the size of this list is $< size(P)$).

MarkIn12 can search, in total, on the whole Q, so that part is $O(n_P \min(n_Q, h_Q \log d_Q))$ (recall **same**). Each element of Q can search on a level of P, and traverse the same level to mark its s-th children, thus we have $O(n_Q(\log d_P + d_P))$, yielding a total of $O(n_P \min(n_Q, h_Q \log d_Q) + n_Q d_P) = O(n \min(n, d + h \log d))$.

MarkIn21, on the other hand, traverses P, thus we have $O(n_Q \min(n_P, h_P \log d_P))$, and each element can mark on a single level of P, thus we have also $O(n_Q d_P)$, yielding $O(n_P + n_Q \min(n_P, d_P + h_P \log d_P)) = O(n \min(n, d + h \log d))$, the same total as MarkIn12.

To see that both MarkIn12 and MarkIn21 can reach $O(n^2)$, consider the following example: $[s]$ $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$ **in** $\{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$.

If P and Q are from the same view, the algorithm behaves better. Since the searches become $O(n \log^2 d)$ (recall "+"), we have that MarkIn12 is $O(n_P \log d_P \log d_Q + n_Q d_P) = O(nd)$. Similarly, MarkIn21 becomes $O(n_Q \log d_Q \log d_P + n_Q d_P) = O(n_Q d_P) = O(nd)$. Then the whole algorithm is $O(nd)$.

If P and Q are from the same view, $[s]$**beginin/endin** should be interpreted as $[s]$**in**.

The same example shown for the general case demonstrates that we can reach $O(n^2)$ even inside a single view.

Merge version: This algorithm requires also marking of selected nodes. We traverse both trees in synchronization. When we find a set of nodes of P included in one of Q, we include the s-ths of them, and then we pass again over the included P-nodes, this time comparing them with the subtree of the Q-node. If, instead, the node of P includes one of Q, we follow the children of the P-node. We finally collect the marked nodes of P.

The algorithm is presented in the next page. It is analyzed as follows.

First, consider that we can traverse both P and Q completely, and that the final deletion of unmarked nodes is $O(n_P)$. Observe that each element of Q is deleted from the problem by doing at most $O(d_P)$ work (when $p_i \subset q_j$), thus the algorithm is $O(n_P + n_Q d_P)$. But also observe that each element of P can be worked on by at most a complete path of Q, thus the algorithm is also $O(n_Q + n_P h_Q)$. Then, the algorithm has the best from both complexities, namely $O(\min(n_P + n_Q d_P, n_Q + n_P h_Q)) = O(n \min(d, h))$.

To see that it can reach $O(n^2)$, consider the same example as for *search-operate*

This analysis is not different if P and Q are from the same view.

The algorithms for $[s]$**beginin** and $[s]$**endin** are quite similar to $[s]$**in**, the only difference being that we should replace (in both places) the condition $p_i \subset q_j$ by $From(p_i) \in q_j$ for **beginin** and by $To(p_i) \in q_j$ for **endin**. The analysis is the same.

If P and Q are from the same view, $[s]$**beginin/endin** should be interpreted as $[s]$**in**.

$[s]$ P **in** Q (*merge* version): Operate (P,Q)

```
Operate (P,Q)

Traverse (P,Q).
Delete unmarked P nodes.

Traverse (P,Q)

While $\max(To(p_i), To(q_j)) < \infty$
    $p_i < q_j$ :  Pass $i$.
    $p_i > q_j$ :  Pass $j$.
    $p_i \subset q_j$ :  Traverse $(\{p_i ...\}, query(q_j))$.   $pos \leftarrow 1$.
              While $p_i \subset q_j$
                  If ($p_i$ is unmarked and $pos \in s$) Mark $p_i$.
                  Pass $i$.   $pos \leftarrow pos + 1$.
    $else$ :     $\{q_j ...\} \leftarrow$ Traverse $(query(p_i), \{q_j ...\})$.  Pass $i$.

Return $\{q_j ...\}$.
```

"with(k)" operator

Search-operate version: We can take each element of Q, search it into P, and increment the annotation of the last P-node including it. The annotation is handled as follows: we store, for each node of P, how many times it has been annotated. Then we traverse P, computing the total number of annotations in its subtrees plus its own annotations, deleting the node if this total is $< k$.

Alternatively, we can take each element of P and determine how many elements of Q it includes, deleting it if includes less than k nodes (by doing it bottom-up, we only count this for nodes without remaining descendants, i.e. as long as a node is determined to pass the test, its ancestors are automatically included). We cannot switch between the two algorithms for the subproblems as before, because they are difficult to combine. Recall that we store the size of each subtree, although we do not detail how we keep it, because it adds uninteresting overhead to the exposition. We also store the counter for each node, which is normally zero.

Observe that overlaps are possible here, since P and Q need not to be from the same view.

Note that we use $parent_P$ in `MarkWith21`. This information is not stored, but easily computed by `Search`.

P **with**(k) Q (*search-operate* version): Operate (P,Q)

```
Operate (P,Q)

If $(size(P)(\log_2(size(Q)) + \min(k, size(Q))) < size(Q)\log_2(size(P)))$ With12 (P,Q).
else With21 (P,Q).

With21 (P,Q)

MarkWith21 $(P, Q, none)$.
Delete (P).
Delete ($P$)

$tot \leftarrow 0$.
For each $p_i$
    $parc \leftarrow annot(p_i) +$ Delete($query(p_i)$).
    $tot \leftarrow tot + parc$.   $annot(p_i) \leftarrow 0$.
    Include $p_i$ only if $parc \geq k$ (it is already a leaf if deleted now).
Return $tot$.
```

```
With12 ($P, Q$)

If $|P| = 1$
    $(Q, b, t) \leftarrow$ Searchi ($node(p), Q$).
    $tot \leftarrow$ With12 ($query(p), \{q_b..q_t\}$).
    If ($tot < k$) $tot \leftarrow$ CheckIncl ($p, \{q_b..q_t\}, k$)
    If ($tot < k$) Delete $p$.
    Return $tot$.
else
    $(Q, b, t) \leftarrow$ Searchi ($\langle From(p_1), To(p_{last})\rangle, Q$).
    Return With12 ($\{p_1..p_{half}\}, \{q_b..q_t\}$) + With12 ($\{p_{half+1}..p_{last}\}, \{q_b..q_t\}$).
```

```
MarkWith21 ($P, Q, top$)

If $|Q| = 1$
    $(P', b, t) \leftarrow$ Search ($node(q), P$).
    If ($P \neq P'$) $top \leftarrow parent_P(P')$.
    MarkWith21 ($\{p'_b..p'_t\}, query(q), top$).
    If ($top \neq none$) $annot(top) \leftarrow annot(top) + 1$.
else
    $(P, b, t) \leftarrow$ Searchi ($\langle From(q_1), To(q_{last})\rangle, P$).
    MarkWith21 ($\{p_b..p_t\}, \{q_1..q_{half}\}$).  MarkWith21 ($\{p_b..p_t\}, \{q_{half+1}..q_{last}\}$).
```

```
CheckIncl ($p, Q, k$)

Restrict $b, t$ until $\langle From(q_b), To(q_t)\rangle \subseteq Segm(p)$.
$n \leftarrow k - (t - b + 1)$.
For each $m \in b..t$, while $n > 0$
    $n \leftarrow n - size(q_m)$.
If ($n \leq 0$) Return $k$.
If $p$ $overlaps$ $q_{b-1}$
    $n \leftarrow n$ − CheckIncl ($p, query(q_{b-1}), n$).  If ($n \leq 0$) Return $k$.
If $p$ $overlaps$ $q_{t+1}$
    $n \leftarrow n$ − CheckIncl ($p, query(q_{t+1}), n$).  If ($n \leq 0$) Return $k$.
Return $k - n$.
```

We analyze this algorithm by considering both alternatives separately.

`With12` makes the search we have already analyzed for **same**, i.e. $O(n_P \min(n_Q, h_Q \log d_Q))$, but also, for each element of P, it can execute `CheckIncl`, which cannot count more than k or n_Q elements, but can traverse a path due to overlaps finding nothing, which adds $\min(n_Q, h_Q \log d_Q)$ to its complexity. Thus, the complexity of `CheckIncl` is $O(\min(n_Q, k + h_Q \log d_Q))$, so we have a total complexity of $O(n_P \min(n_Q, k + h_Q \log d_Q)) = O(n \min(n, k + h \log d))$.

To see that it can reach $O(n^2)$, consider $\{\langle n, 2n\rangle, \langle n, 2n-1\rangle, ..., \langle n, n+1\rangle\}$ **with**(1) $\{\langle 1, n\rangle, \langle 2, n\rangle, ..., \langle n, n\rangle\}$ (which is also an $O(n)$ example for `CheckIncl`).

`With21` is much simpler, since it just makes the traversal we have analyzed for **same**, and collects P to select the properly marked nodes, i.e. $O(n_P + n_Q \min(n_P, h_P \log d_P)) = O(n \min(n, h \log d))$.

To show that it can reach $O(n^2)$, consider the same example as for **same**.

We can use just use `With21` if faced to the problem of having better worst case.

If P and Q are from the same view, the algorithm behaves better.

First, (as we saw for "+"), the searches into a set are $O(n \log^2 d)$ in total. Second, we show that now `CheckIncl` can be made $O(\log d_Q)$.

Since there are not overlaps, `CheckIncl` is $O(\log d_Q + \min(k, d_Q))$. But we could form an (implicit) binary tree on the levels of Q, where nodes represent ranges $1..d, 1..(d/2), (d/2+1)..d, ...$, and store at each node of the binary tree the total size of the interval it represents. In this case, the counting of nodes in a call to `CheckIncl` is also $O(\log d_Q)$, regardless of k, and thus the whole `CheckIncl` becomes $O(\log d_Q)$.

This way, `With12` becomes $O(n_P(\log d_P \log d_Q + \log d_Q)) = O(n_P \log d_P \log d_Q) = O(n \log^2 d)$. `With21` becomes $O(n_P + n_Q \log d_Q \log d_P) = O(n \log^2 d)$, and the whole algorithm is $O(n \log^2 d)$.

As for "+", we have not found a $O(n \log^2 n)$ example. An $O(n \log n)$ example for `With12` is $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n\rangle\}$ **with**$(n+1)\{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$. An $O(n \log n)$ example for `With21` is $\{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$ **with**(1) $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n\rangle\}$.

Merge version: We traverse both trees synchronously, to determine, for each node of Q, the last node of P containing it, then incrementing its annotation.

P **with**(k) Q (*merge* version): Operate (P, Q)

```
Operate ($P, Q$)

Annotate ($P, Q, none$).
Delete ($P$).

Annotate ($P, Q, last$)

While $To(q_j) < (last \neq none \ ? \ To(last) + 1 \ : \ \infty)$
        $p_i < q_j$ :     Pass $i$.
        $p_i > q_j$ :     If ($last \neq none$) $annot(last) \leftarrow annot(last) + 1 + size(q_j)$.
                          Pass $j$.
        $p_i \supset q_j$ :  $\{q_j...\} \leftarrow$ Annotate ($query(p_i), \{q_j...\}, p_i).restq$.
        $else$ :          If ($last \neq none$) $annot(last) \leftarrow annot(last) + 1$.
                          $\{p_i...\} \leftarrow$ Annotate ($\{p_i...\}, query(q_j), last).restp$.   Pass $j$.

Return ($restp = \{p_i...\}, restq = \{q_j...\}$).
```

This algorithm is $O(n_P + n_Q) = O(n)$, what can be seen by considering again that each comparison leaves at least one element out of the problem, and that the collection of properly marked nodes is $O(n_P)$.

To see that it can reach $O(n)$, we use the same example as for "+".

This algorithm is not better if P and Q are from the same view.

"withbegin/withend(k)" operators

These operations are similar to **with**, the necessary modifications follow.

If P and Q are from the same view, **withbegin/withend**(k) should be interpreted as **with**(k).

Search-operate version: We have to add in `CheckIncl` an instruction just after the definition of n, to say `If` $(p_i\ \mathit{overlaps}\ q_{t+1}/q_{b-1})$ $n \leftarrow n-1$. On the other hand, `MarkWith21` must be modified to be as follows

```
MarkWith21 $(P, Q, top)$

If $|Q| = 1$
   $(P', b, t) \leftarrow$ Searchi $(node(q), P)$.  If $(P \neq P')$ $top \leftarrow parent_P(p'_b)$.
   MarkWith21 $(\{p'_b..p'_t\}, query(q), top)$.
   $(P', b, t) \leftarrow$ Searchi $(\langle From/To(q), From/To(q)\rangle, \{p'_b, p'_t\})$.
   If $(P \neq P')$ $top \leftarrow parent_P(P')$.
   If $(top \neq none)$ $annot(top) \leftarrow annot(top) + 1$.
else
   $(P', b, t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$ If $(P \neq P')$ $top \leftarrow parent_P(p'_b)$.
   MarkWith21 $(\{p_b..p_t\}, \{q_1..q_{half}\}, top)$.  MarkWith21 $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\}, top)$.
```

The analysis remains the same.

Merge version: The only difference is that we should change the segment whose annotation is incremented in the *else* condition. In **withbegin**, if $From(q_j) \in p_i$ we increment the annotation of p_i, not *last*. In **withend**, if $To(q_j) \in p_i$ we increment the annotation of p_i, not *last*. The analysis is the same.

"[s]child" operator

Search-operate version: We can take each element of P and search its (view) parent in Q, marking the P-node if it is s-th child of its view parent and that view parent is in Q; or we can take each element of Q and search its view children in P, marking its s-th view children. We can switch the algorithms for subproblems.

Observe that overlaps are not possible here, since P and Q must be from the same view.

[s] P **child** Q (*search-operate* version): Operate (P, Q)

```
Operate $(P, Q)$

Child $(P, Q, none)$.
Delete unmarked nodes from $P$.

Child $(P, Q, par)$

If $(size(P)\log_2(size(Q)) < 2size(Q)\log_2(size(P)))$ Child12 $(P, Q, par)$.
else Child21 $(P, Q)$.
```

```
Child12 $(P, Q, par)$

If $|P| = 1$
   $(Q', b, t) \leftarrow$ Searchi $(node(p), Q)$.  If $(Q \neq Q')$ $par \leftarrow parent_Q(q'_b)$.
   If $p$ is unmarked $\wedge$ $par \neq none$ $\wedge$ $parent_{view}(p) = par$ $\wedge$ $pos_{view}(p) \in s$
      Mark $p$.
   Child $(query(p), \{q'_b..q'_t\}, par)$.
else
   $(Q', b, t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, Q)$.  If $(Q \neq Q')$ $par \leftarrow parent_Q(q'_b)$.
   Child $(\{p_1..p_{half}\}, \{q'_b..q'_t\}, par)$.  Child $(\{p_{half+1}..p_{last}\}, \{q'_b..q'_t\}, par)$.
```

```
Child21 $(P,Q)$

If $|Q|=1$
    $(P,b,t) \leftarrow$ Searchi $(node(q),P)$.
    For each $m \in b..t$
        If $(parent_{view}(p_m)=q \ \wedge \ pos_{view}(p_m) \in s)$ Mark $p_m$.
    Child $(\{p_b..p_t\}, query(q), q)$.
else
    $(P,b,t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
    Child $(\{p_b..p_t\}, \{q_1..q_{half}\}, par)$.   Child $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\}, par)$.
```

We analyze this algorithm by considering each alternative. The final deletion of unmarked nodes is $O(n_P)$.

`Child12` makes the traversal we analyzed for "+", thus it is $O(n_P \log d_Q..n_P \log d_P \log d_Q) = O(n \log d..n \log^2 d)$.

To see that it can reach $O(n \log n)$, consider $[s]$ $\{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$ **child** $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$.

`Child21` does the same, but it may have to traverse linearly a level of P for each node of Q, so it is $O(n_P + n_Q d_P) = O(nd)$.

To see that it can reach $O(n^2)$, consider $[s]$ $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$ **child** $\{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$.

Again, if we are faced to the problem of having low worst case behavior, we simply select `Child12` always.

Merge version: We traverse both trees in synchronization, recalling the last Q-node including the current P elements (par). When we find a P-node which descends from par, we test if it is an s-th view child of par, in which case we collect it.

$[s]$ P $\mathbf{child}_{par}$ Q (*merge* version): Operate (P,Q,par).result

```
Operate $(P,Q,par)$

$res \leftarrow \lambda$.
While $To(p_i) < (par = none \ ? \ \infty \ : \ To(par)+1)$
    $p_i < q_j$ :        If $par \neq none \ \wedge \ parent_{view}(p_i) = par \ \wedge \ pos_{view}(p_i) \in s$
                             $res \leftarrow res : (node(p_i), \lambda)$.
                         Pass $i$.
    $p_i > q_j$ :        Pass $j$.
    $p_i \supseteq q_j$ :  $Op \leftarrow$ Operate $(query(p_i), \{q_j...\}, par)$.
                         If $par \neq none \ \wedge \ parent_{view}(p_i) = par \ \wedge \ pos_{view}(p_i) \in s$
                             $res \leftarrow res : (node(p_i), Op.result)$.
                         else $res \leftarrow res : Op.result$.
                         $\{q_j...\} \leftarrow Op.restq$.   Pass $i$.
    $p_i \subset q_j$ :    $Op \leftarrow$ Operate $(\{p_i...\}, query(q_j), node(q_j))$.
                         $res \leftarrow res : Op.result$.   $\{p_i...\} \leftarrow Op.restp$.   Pass $j$.
Return $(result = res, restp = \{p_i...\}, restq = \{q_j...\})$.
```

This algorithm is $O(n_P + n_Q) = O(n)$, what can be seen by considering again that each comparison leaves one element out of the problem.

To show that it can reach $O(n)$, we use the same example of "+".

"parent(k)" operator

Search-operate version: We can take each element of P and search it in Q, marking the P-node if it has k or more of its children in Q. Or we can take each element of Q and search its parent in P, incrementing its annotation. Then, we delete the P-nodes whose annotation is less than k. We can switch algorithms for subproblems.

Observe that overlaps are not possible here, since P and Q must be from the same view.

P **parent**(k) Q (*search-operate* version): Operate (P, Q)

```
Operate $(P,Q)$

Parent $(P,Q)$.
Delete from $P$ those nodes whose $annot < k$.

Parent $(P,Q)$

If $(2 size(P)\log_2(size(Q)) < size(Q)\log_2(size(P)))$ Parent12 $(P,Q)$.
else For each $p_i$ Parent21 $(query(p_i), Q, p_i)$.
```

```
Parent12 $(P,Q)$

If $|P| = 1$
   $(Q,b,t) \leftarrow$ Searchi $(node(p), Q)$.
   Parent $(query(p), \{q_b..q_t\})$.
   If $t - b + 1 \geq k$
      For each $m \in b..t$, while $annot(p) < k$
         If $(p = parent_{view}(q_m))$ $annot(p) \leftarrow annot(p) + 1$.
else
   $(Q,b,t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, Q)$.
   Parent $(\{p_1..p_{half}\}, \{q_b..q_t\})$.  Parent $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\})$.
```

```
Parent21 $(P,Q,par)$

If $|Q| = 1$
   $(P',b,t) \leftarrow$ Searchi $(node(q), P)$.  If $(p \neq P')$ $par \leftarrow parent_P(p'_b)$.
   Parent $(\{p'_b..p'_t\}, query(q))$.
   If $(par = parent_{view}(q))$ $annot(par) \leftarrow annot(par) + 1$.
else
   $(P,b,t) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, P)$.
   Parent $(\{p_b..p_t\}, \{q_1..q_{half}\})$.  Parent $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\})$.
```

We analyze this algorithm by considering each alternative. The final deletion of unmarked nodes is $O(n_P)$.

`Parent12` makes the traversal we analyzed for "+", i.e. $O(n_P \log d_P \log d_Q)$, but it can also traverse, for each node of P, a level of Q, thus it is $O(n_P d_Q) = O(nd)$ in total.

To see that it can reach $O(n^2)$, consider $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$ **parent**(1) $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$, if neither node of P is parent of a node of Q.

`Parent21` makes the same traversal, but nothing else, so it is $O(n_P + (n_Q \log d_P..n_Q \log dP \log d_Q)) = O(n \log d..n \log^2 d)$.

To see that it can reach $O(n \log n)$, consider $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$ **parent**(1) $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$.

If faced to the problem of having best worst-case behavior, we can use only `Parent21`.

Merge version: We traverse both trees in synchronization. When we find in P the parent of a node of Q, we increment its annotation. Finally, we delete the P-nodes having $< k$ children.

P **parent**(k) Q (*merge* version): Operate (P,Q)

```
Operate $(P,Q)$

Annotate $(P,Q,none)$.
Delete $P$-nodes whose $annot < k$.
```

```
Annotate $(P,Q,par)$

While $To(q_j) < (par = none\ ?\ \infty\ :\ To(par)+1)$

    $p_i < q_j$ :        Pass $i$.
    $p_i > q_j$ :        If $(par \neq none\ \wedge\ parent_{view}(q_j) = par)$ $annot(par) \leftarrow annot(par)+1$.
                         Pass $j$.
    $p_i \supset q_j$ :  $\{q_j ...\} \leftarrow$ Annotate $(query(p_i), \{q_j ...\}, p_i)$.$restq$.
                         Pass $i$.
    $p_i \subseteq q_j$ : If $(par \neq none\ \wedge\ parent_{view}(q_j) = par)$ $annot(par) \leftarrow annot(par)+1$.
                         $\{p_i ...\} \leftarrow$ Annotate $(\{p_i ...\}, query(q_j), none)$.$restp$.
                         Pass $j$.

Return $(restp = \{p_i ...\}, restq = \{q_j ...\})$.
```

This algorithm is $O(n_P + n_Q) = O(n)$, which can be seen by considering that each time `Annotate` is recursively invoked, the part of the problem it has traversed is never worked on again, and that at each iteration, at least one element is deleted from the problem. Finally, the deletion of unmarked nodes is $O(n_P)$.

To see that it can reach $O(n)$, we use the same example of "+".

"before/after$(k)(C)$" operators

Search-operate version: We begin by solving **before**. We can take each element of P and search into Q the adjacent corresponding range (which is passing the P-node, up to k points to the right, but can also be limited by the minimal C-node containing the P-node), marking the P-node if we find something in Q beginning within the range and with the same minimal C including node (`Findf`). Or we can take each element of Q, compute its adjacent corresponding range (which is before the Q-node), and search it into P, marking the corresponding nodes in P. Later, we delete the unmarked P-nodes. We cannot switch algorithms at subproblems, since they are difficult to combine. If both minimal C including nodes do not exist, they are *none* and are considered to be equal and to include everything.

Observe that overlaps are possible here, since P, Q and C can be from different views.

We use an auxiliary procedure in the "12" version: `Findf`, which receives the range where to search, the top-level Q candidates, and the minimal C including the P-node, and searches into the Q candidates for someone included in the same minimal C-node. Except for the extremes, the rest of the top-level nodes are guaranteed to be included in the C-node. One of the (perhaps overlapping) extremes can also be used to make the P-node classify . Note that if a non-overlapping top-level Q-node is included in another C-node (which is then included in the original C), then its subtree is also included in that new C. The only cases in which searching into subtrees of Q is necessary is in overlapping extremes.

P **before**(k) Q (C) (*search-operate* version): Operate (P, Q, C)

```
Operate $(P, Q, C)$

If $(size(P)(\log_2(size(C)) + 2\log_2(size(Q))) <$
     $size(Q)(\log_2(size(C)) + 2\log_2(size(P))) + size(Q))$
    Before12 $(P, Q, C)$.
else Before21 $(P, Q, C)$.

Before21 $(P, Q, C)$

MarkBefore21 $(P, Q, C)$.
Delete unmarked nodes.
```

```
Before12 $(P, Q, C)$

If $|P| = 1$
    $(C', db, dt) \leftarrow$ Searchi $(node(p), C)$.
    $s \leftarrow \langle From(p), To(p) + k\rangle \ \cap \ parent_C(C')$.
    $(Q, b, t) \leftarrow$ Search $(s, Q)$.
    Before12 $(query(p), \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
    If $(\neg$ Findf $(\langle To(p) + 1, To(s)\rangle, \{q_b..q_t\}, parent_C(c'_{db})))$ $P \leftarrow query(P)$.
else
    $(C', db, dt) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, C)$.
    $s \leftarrow \langle From(p_1), To(p_{last}) + k\rangle \ \cap \ parent_C(c'_{db})$.
    $(Q, b, t) \leftarrow$ Search $(s, Q)$.
    Before12 $(\{p_1..p_{half}\}, \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
    Before12 $(\{p_{half+1}..p_{last}\}, \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
```

```
Findf $(s, Q, c)$

$(Q, b0, t0) \leftarrow$ Search $(s, Q)$.
$b \leftarrow (q_{b0}\ overlaps\ s\ ?\ b0 + 1\ :\ b0)$.
$t \leftarrow (q_{t0}\ overlaps\ s\ ?\ t0 - 1\ :\ t0)$.
For each $j \in b..t$
    Restrict $db, dt$ while $query(c)_{db-1} < q_j < query(c)_{dt+1}$.
    If $(q_j \not\subseteq query(c)_b)$ Return $true$.
If $t0 \neq t$
    Restrict $db, dt$ while $query(c)_{db-1} < q_{t0} < query(c)_{dt+1}$.
    If $q_{t0} \subseteq c$
        If $(q_{t0} \not\subseteq query(c)_{db})$ Return $true$.
    else If (Findf $(s, query(q_{t0}), (node(c), \{c_{db}..c_{dt}\})))$ Return $true$.
If $b0 \neq b$
    Restrict $db, dt$ while $query(c)_{db-1} < q_{b0} < query(c)_{dt+1}$.
    If $(q_{b0} \not\subseteq query(c)_{db}\ \wedge$ Findf $(s, query(q_{b0}), (node(c), \{c_{db}..c_{dt}\})))$ Return $true$.
Return $false$.
```

```
MarkBefore21 $(P, Q, C)$

If $|Q| = 1$
    $(C', db, dt) \leftarrow$ Searchi $(node(q), C)$.
    $s \leftarrow \langle From(q) - k, To(q)\rangle \;\cap\; parent_C(c'_{db})$.
    $(P, b, t) \leftarrow$ Search $(s, P)$.
    MarkBefore21 $(\{p_b..p_t\}, query(q), \{c'_{db}..c'_{dt}\})$.
    MarkInside $(\langle From(s), From(q) - 1\rangle, \{p_b..p_t\}, parent_C(c'_{db}))$.
else
    $(C', db, dt) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, C)$.
    $s \leftarrow \langle From(q_1) - k, To(q_{last})\rangle \;\cap\; parent_C(c'_{db})$.
    $(P, b, t) \leftarrow$ Search $(s, P)$.
    MarkBefore21 $(\{p_b..p_t\}, \{q_1..q_{half}\}, \{c'_{db}..c'_{dt}\})$.
    MarkBefore21 $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\}, \{c'_{db}..c'_{dt}\})$.
```

We use two auxiliary procedures here. `MarkInside` receives a range s, a top-level P-nodes candidates for marking, and the minimal C containing the marking Q-node. The P-nodes overlapping the range are traversed recursively to find more candidates (one of the overlapping extremes can also be marked); and those nodes wholly included in the range are sent to `MarkFull`. This procedure uses a second type of mark, `mark2`. It traverses the whole subtree of the P-nodes, marking them, and stopping when a P-node descends from a C-node included in the minimal C containing the Q-node. This `mark2` has another property that avoids repeating work: if a node has been `marked2` once, there is no benefit in entering it again to mark it with other Q-node. The reason for this is that `MarkBefore21` proceeds bottom-up and left-to-right. That guarantees that any P-node subtree is `MarkFull`'ed first with the best Q-node to do it, since it is done only if the P-node (and hence its subtree) is fully contained in the C-node, and with the minimal possible C-node; since the C-node containing both the P-subtree and a higher or farther Q-node than the first one is surely equal or larger than the first one. Observe that this is because `MarkBefore21` marks P-nodes to its left, the corresponding `MarkAfter21` should invoke `MarkInside` also bottom-up but rigth-to-left.

```
MarkInside $(s, P, c)$

$(P, b, t) \leftarrow$ Search $(s, P)$.
$b0 \leftarrow (b < t - 1 \;\wedge\; p_b \; overlaps \; s \;?\; b + 1 \;:\; b)$.
$t0 \leftarrow (p_t \; overlaps \; s \;?\; t - 1 \;:\; t)$.
MarkFull $(\{p_{b0}..p_{t0}\}, query(c))$.
If $b \neq b0 \;\wedge\; p_b$ is unmarked2
    Restrict $b', t'$ while $query(c)_{b'-1} < p_b < query(c)_{t'+1}$.
    If $(c \supseteq p_b \not\subseteq query(c)_{b'})$ Mark $p_b$.
    MarkInside $(s, query(p_b), (node(c), \{query(c)_{b'}..query(c)_{t'}\}))$.
If $t \neq t0 \;\wedge\; p_t$ is unmarked2
    Restrict $b', t'$ while $query(c)_{b'-1} < p_t < query(c)_{t'+1}$.
    MarkInside $(s, query(p_t), (node(c), \{query(c)_{b'}..query(c)_{t'}\}))$.
```

```
MarkFull $(P, C)$

If $|P| = 1$
    If $p$ is unmarked2
        Restrict $b, t$ while $c_{b-1} < p < c_{t+1}$.
        If $(p \not\subseteq c_b)$ Mark2 $p$.  MarkFull $(query(p), \{c_b..c_t\})$
else
    $(C, b, t) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last} + k)\rangle, C)$.
    MarkFull $(\{p_1..p_{half}\}, \{c_b..c_t\})$.  MarkFull $(\{p_{half+1}..p_{last}\}, \{c_b..c_t\})$.
```

We analyze this algorithm by considering each alternative.

`Before12` traverses C as we analyzed for **same**, so this part is $O(n_P \min(n_C, h_C \log d_C))$. Although it does something somewhat different for Q (we search not for the nodes, but for their extension by s), we have anyway $O(n_P \min(n_Q, h_Q \log d_Q))$ for that part. Finally, `Findf`, which is called for all P-nodes, can traverse a complete path of the tree, and at each level we traverse we search its d_Q nodes into another level (of C). This leads us to $O(\min(n_Q, h_Q d_Q) \log d_C)$ for `Findf`.

But note that we can replace the loop `For` $j \in b..t$ by a binary tree strategy similar to artificial nodes, searching for $\langle From(q_b), To(q_t)\rangle$, then for $\langle From(q_b), To(q_{(b+t)/2})\rangle$ and $\langle From(q_{(b+t)/2+1}), To(q_t)\rangle$, etc. The total complexity in this case is $O(d_Q + d_C)$ instead of $O(d_Q \log d_C)$. To see this, consider first that each partition of the Q level cuts the C level by half, then we have $\log d + 2\log(d/2) + ... + 2^i \log(d/2^i) + ... + d \log 1$, which is $O(d)$. We can also show that the worst case occurs when we partition by half the searched level, since if the partition is with proportion α we have

$$\sum_{i=0}^{\log_2 d} \sum_{k=0}^{i} \binom{i}{k} \log_2\left(\alpha^k (1-\alpha)^{j-k}\right) = (2d-1)\log_2 d + \log_2(\alpha(1-\alpha))(d \log_2 d - d + 1)$$

which is maximized when $\alpha = 1/2$, to yield $2d - \log_2 d - 1$.

Therefore, the complexity of `Findf` can be reduced to $\min(n_Q \log d_C, h_Q(d_C + d_Q))$ (it cannot be worse than $n_Q \log d_C$, since the search done with the binary tree approach can never be more than if each node searched in a whole level). This way, we have a total complexity of $O(n_P(\min(n_C, h_C \log\ d_C) + \min(n_Q, h_Q \log d_Q) + \min(n_Q \log d_C, h_Q(d_C + d_Q)))) = O(n \min(n \log d, hd))$.

To see that it can reach $O(n^2 \log n)$, consider $\{\langle 1, 2n\rangle, \langle 2, 2n\rangle, ..., \langle n, 2n\rangle\}$ **before**$(n+1)$ $\{\langle 1, 3n\rangle, \langle 1, 3n-1\rangle, ..., \langle 1, 2n+1\rangle, \langle 2n+1, 2n+1\rangle\}$ $(\{\langle 1, 3n+1\rangle, \langle n, n\rangle, \langle n+1, n+1\rangle, ..., \langle 3n, 3n\rangle\})$ which is also an $O(n \log n)$ example for `Findf`.

`MarkBefore21` uses marking, so we must add $O(n_P)$. It traverses C as we analyzed for **same**, so that part is $O(n_Q \min(n_C, h_C \log d_C))$. By the same reasons as before, the search into P is also $O(n_Q \min(n_P, h_P \log d_P))$. `MarkInside` is called for each Q-node, and it can traverse a complete path, searching into a level of C each time, thus it is $O(h_P \log d_C)$ if we do not count the calls to `MarkFull`, which can be invoked at each level. `MarkFull` can, in whole, traverse the complete tree (because of `mark2`), searching into a level of C for each element, thus it is $O(n_P \log d_C)$. Therefore, we have a total complexity of $O(n_Q(\min(n_C, h_C \log d_C) + \min(n_P, h_P \log d_P) + h_P \log d_C) + n_P \log d_C) = O(nh \log d)$.

To see that it can reach $O(n^2 \log n)$, consider $\{\langle n, 3n\rangle, \langle n, 3n-1\rangle, ..., \langle n, 2n+1\rangle\}$ **before**$(n+1)$ $\{\langle n, 2n\rangle, \langle n, 2n-1\rangle, ..., \langle n, n\rangle\}$ $(\{\langle 1, 3n+1\rangle, \langle n, n\rangle, \langle n+1, n+1\rangle, ..., \langle 3n, 3n\rangle\})$.

If faced to the problem of having better worst case behavior, we prefer to use only `Before21`.

If P, Q and C are from the same view, the algorithm behaves a little better. The searches into C are now $O(n_P \log d_P \log d_C)$ in total, but the rest of the procedures involve s, that can still cause overlaps, and then their complexity remains the same. This way, the total complexity of `Before12` becomes $O(n_P(\log d_P \log d_C + \min(n_Q, h_Q \log d_Q) + \min(n_Q \log d_C, h_Q(d_C + d_Q)))) = O(n \min(n \log d, hd))$, a little change. The total complexity of `Before21` becomes $O(n_Q(\log d_P \log d_C + \min(n_P, h_P \log d_P) + h_P \log d_C) + n_P \log d_C) = O(nh \log d)$, also a little change.

The same example given for the general case can show that even inside the same view the algorithm is $O(n^2 \log n)$.

The idea for **after**$(k)(C)$ is the same as for **before**, only some details differ. The complexity is also the same.

Merge version: We begin by solving **before**. We traverse both trees in synchronization, determining for each node of P if there is a node of Q under the same minimal C-node, after the P-node, at a distance $\leq k$.

P **before**(k) Q (C) (*merge* version): Operate (P, Q)

```
Operate $(P,Q)$

Traverse $(P,Q)$.
Delete unmarked elements from $P$.

Traverse $(P,Q)$

While $To(p_i) < \infty$

    { $p_i > q_j$ :  Pass $j$.
    { $else$ :       $(c, lc) \leftarrow$ SearchC $(node(p_i))$.
    {                $s \leftarrow \langle To(p_i)+1, To(p_i)+k \rangle \ \cap \ c$.
    {                $\{q_j ...\} \leftarrow$ Traverse $(query(p_i), \{q_j ...\})$.
    {                While $(To(q_j) < From(s))$ Pass $j$.
    {                If (Determ $(\{q_j ...\}, s, To(c), lc).tomark$) Mark $(p_i)$.
    {                Pass i.

Return $(\{q_j ...\})$.
```

`SearchC` searches for the minimal element of C including its argument. To achieve total $O(nh)$ behavior for `SearchC`, it uses a global array G where the current position of the search at each level of C is kept, to traverse at most one path at each invocation (note that P is traversed in depth-first order).

```
SearchC $(p_i)$

Call $U$ a special segment containing all.
Call $G$ the global array, $L$ the current level.
Initially $G_0 = U, G_1 = c_1, L = 1$.

While $(p_i \not\subseteq G_{L-1})$ $L \leftarrow L-1$.
Repeat
   While $(G_L \neq end \ \wedge \ To(G_L) < From(p_i))$ Pass $G_L$.   $G_{L+1} \leftarrow none$.
   If $G_L \neq end \ \wedge \ p_i \subseteq G_L$
      $L \leftarrow L+1$.
      If $(G_L = none)$ $G_L \leftarrow first(query(G_{L-1}))$.   $G_{L+1} \leftarrow none$.
   else Return $(G_{L-1}, G_L)$.
```

`Determ` searches for an element in Q included in the same minimal C node, and at the proper direction and distance.

```
Determ $(\{q_j ...\}, s, qlim, \{c_i\})$

If $From(q_j) < From(s)$
   $(tomark, \{c_i ...\}) \leftarrow$ Determ$(query(q_j), s, qlim, \{c_i ...\})$.
   If $(tomark)$ Return $(true, \lambda)$ else Pass $j$.

While $From(q_j) \leq To(s) \ \wedge To(q_j) \leq qlim$

   { $c_i < q_j$ :       Pass $i$.
   { $c_i \supseteq q_j$ :  Pass $j$.
   { $else$ :            Return $(true, \lambda)$.

If $(From(q_j) \leq To(s))$ Return Determ$(query(q_j), s, qlim, \{c_i ...\})$.
Return $(false, \{c_i ...\})$.
```

This algorithm is analyzed as follows.

First, observe that throughout the whole execution, `SearchC` works $O(n_C h_C)$ time, since the total number of times that the "`Pass` G_L" instruction is executed is $O(n_C)$, because a node passed that way is never reconsidered again (due to the depth-first traversal of P); and on the other hand, we can traverse the whole tree up and down at each call, thus achieving a total $O(nh)$ behavior.

To see that `SearchC` is no better than $O(n^2)$, consider $C = \{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$, $P = \{\langle n, 2n+1\rangle, \langle n, n\rangle, \langle n+1, 2n+1\rangle, \langle n+1, n+1\rangle, \langle n+2, 2n+1\rangle, ...\}$.

`Determ` is $O(\min(n_Q, d_Q h_Q))$, since it works $O(d_Q)$ at each iteration, and only the first time it can get into both recursive calls. Since at each iteration it descends one level in Q, we have it is $O(d_Q h_Q)$. On the other hand, it cannot traverse more than the whole Q, so it is $O(\min(n_Q, d_Q h_Q))$. Note that it traverses also a level of C, but this level is traversed only once along the whole path, so it adds $O(d_C)$ at each invocation from outside.

To see that `Determ` can reach $O(n)$, consider `Determ` `(`$\{\langle 1, n\rangle, \langle 2, n\rangle, ..., \langle n, n\rangle\}, \langle n, 2n\rangle, \{c_i ...\}$`)`.

The number of times q_j is advanced without progressing in P is $O(d_Q)$ in total (i.e. the top-level of Q). Finally, each p_i is left out of the problem by executing `Determ`, i.e. this part is $O(n_P(d_C + \min(n_Q, d_Q h_Q)))$.

Therefore, the total complexity is $O(n_C h_C + n_P(d_C + \min(n_Q, d_Q h_Q))) = O(n \min(n, dh))$.

To see that it can reach $O(n^2)$, consider the following example: $\{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$ **before**$(n+1)$ $\{\langle n+1, n+1\rangle, \langle n+2, n+2\rangle, ..., \langle 2n, 2n\rangle\}$ $(\{\langle 1, 2n\rangle, \langle n+1, 2n\rangle\})$.

If P, Q and C are from the same view, the algorithm behaves a little better. The reason is that now `SearchC` makes a single traversal on C along the whole execution, since the worst that can happen is that C is wholly traversed in depth-first order. Not existing overlaps, it is not possible to repeat a path. The whole complexity is then $O(n_C + n_P(d_C + \min(n_Q, d_Q h_Q))) = O(n \min(n, dh))$, a little improvement.

The same example of the general case shows that we can reach $O(n^2)$ even inside a single view.

The idea for **after**$(k)(C)$ is the same as for **before**, only some details differ. The complexity is also the same.

"before/after(C)" operators

Search-operate version: We begin by solving **before**. We can take each element of Q, compute its adjacent corresponding range (which is before the Q-node, limited by the minimal C-node including the Q-node), and search it in P, marking the node in P which is nearest the one of Q. Later, we delete the unmarked P-nodes. Or we can take each element of P and search in Q the adjacent corresponding segment (which is passing the P-node), selecting from Q the node which is nearest to the P-node. Then, we search the Q-node back in P, to determine the P-node that that Q-node should mark. Since the best candidate for marking a P-node is its nearest Q-node, and each P-node gets its best opportunity to be marked, it suffices to mark only on Q-nodes selected as the best for each P-node (note that if the "best" Q-node for a given P-node does not mark it, then neither will, since that means that there is another P-node which is preferred by the Q-node, and that one will be preferred by any farther node). We also mark the already used nodes of Q to avoid repeating work.

We cannot switch algorithms at subproblems, since they are difficult to combine. Observe that overlaps are possible here, since P, Q and C can be from different views.

P **before** Q (C) (*search-operate* version): Operate (P, Q, C)

```
Operate $(P,Q)$

If $(2size(P)\log_2(size(Q)) < size(Q)\log_2(size(P)))$ BeforeMin12 $(P,Q,C)$.
else MarkBeforeMin21 $(P,Q,C)$.
Delete unmarked nodes from $P$.

BeforeMin12 $(P,Q,C)$

MarkBeforeMin12 $(P,P,Q,C)$.
Unmark nodes of $Q$.
```

```
MarkBeforeMin12 $(Pr,P,Q,C)$

If $|P| = 1$
   $(C', db, dt) \leftarrow$ Searchi $(node(p), C)$.
   $s \leftarrow \langle From(p), \infty\rangle \;\cap\; parent_C(c'_{db})$.
   $(Q,b,t) \leftarrow$ Search $(s,Q)$.   $(Pr,b',t') \leftarrow$ Search $(s,Pr)$.
   MarkBeforeMin12 $(\{Pr_{b'}..Pr_{t'}\}, query(p), \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
   If $p$ is unmarked
      $B \leftarrow$ SearchMinf $(\langle To(p)+1, To(s)\rangle, \{q_b..q_t\}, parent_C(c'db))$.
      If $B \neq none \;\wedge\; B$ is unmarked
         Mark $B$.   $s \leftarrow \langle From(s)-1, From(B)-1\rangle$.
         $r \leftarrow$ SearchMint $(s, \{Pr_{b'}..Pr_{t'}\}, parent_C(c'_{db}))$.
         If $(r \neq none)$ Mark $r$.
else
   $(C', db, dt) \leftarrow$ Searchi $(\langle From(p_1), To(p_{last})\rangle, C)$.
   $(Q,b,t) \leftarrow$ Search $(\langle From(p_1), \infty\rangle, Q)$.
   $(Pr,b',t') \leftarrow$ Search $(\langle From(p_1), \infty\rangle, Pr)$.
   MarkBeforeMin12 $(\{Pr_{b'}..Pr_{t'}\}, \{p_1..p_{half}\}, \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
   MarkBeforeMin12 $(\{Pr_{b'}..Pr_{t'}\}, \{p_{half+1}..p_{last}\}, \{q_b..q_t\}, \{c'_{db}..c'_{dt}\})$.
```

```
MarkBeforeMin21 $(P,Q,C)$

If $|Q| = 1$
   $(C', db, dt) \leftarrow$ Searchi $(node(q), C)$.
   $s \leftarrow \langle 1, To(q)\rangle \;\cap\; parent_C(c'_{db})$.
   $(P,b,t) \leftarrow$ Search $(s,P)$.
   MarkBeforeMin21 $(\{p_b..p_t\}, query(q), \{c'_{db}..c'_{dt}\})$.
   $r \leftarrow$ SearchMint $(\langle From(s), From(q)-1\rangle, \{p_b..p_t\}, parent_C('_{db}))$.
   If $(r \neq none)$ Mark $r$.
else
   $(C', db, dt) \leftarrow$ Searchi $(\langle From(q_1), To(q_{last})\rangle, C)$.
   $(P,b,t) \leftarrow$ Search $(\langle 1, To(q_{last})\rangle, P)$.
   MarkBeforeMin21 $(\{p_b..p_t\}, \{q_1..q_{half}\}, \{c'_{db}..c'_{dt}\})$.
   MarkBeforeMin21 $(\{p_b..p_t\}, \{q_{half+1}..q_{last}\}, \{c'_{db}..c'_{dt}\})$.
```

```
SearchMinf/SearchMint ($s, P, c$)

$(P, b, t) \leftarrow$ Search ($s, P$).
If $p_{b/t}$ $overlaps$ $s$
   Restrict $b', t'$ while $query(c)_{b'-1} < p_{b/t} < query(c)_{t'+1}$.
   If $p_{b/t} \not\subseteq query(c)_{b'}$
      $r \leftarrow$ SearchMinf/SearchMint ($s, query(p_{b/t}), (node(c), \{query(c)_{b'}..query(c)_{t'}\})$).
      If ($r \neq none$) Return $r$.
      $b \leftarrow b+1$ / $t \leftarrow t-1$.
For each $i \in b..t/t..b$
   Restrict $b', t'$ while $query(c)_{b'-1} < p_i < query(c)_{t'+1}$.
   If ($p_i \not\subseteq query(c)_{b'}$) Return $p_i$.
Return $none$.
```

We analyze this algorithm by considering each alternative. The final deletion of unmarked nodes is $O(n_P)$.

`MarkBeforeMin21` traverses C, in the form we analyzed for **same**, i.e. $O(n_Q \min(n_C, h_C \log d_C))$. It also traverses P, and as explained for **before**$(k)(C)$, it is also $O(n_Q \min(n_P, h_P \log d_P))$. Finally, it invokes `SearchMint` for each node. This routine can traverse a complete path, and for each level it can traverse it wholly; it also searches in a level of C for each element it traverses, so it is $O(\min(n_P, h_P d_P) \log d_C)$. But we can modify it with the same technique used for `Findf`, with a binary tree, to achieve $O(\min(n_P \log d_C, h_P(d_P + d_C)))$. This leads to a total complexity of $O(n_P + n_Q(\min(n_C, h_C \log d_C) + \min(n_P \log d_C, h_P(d_P + d_C)))) = O(n \min(n \log d, hd))$.

To see that it can reach $O(n^2 \log n)$, we use the same example as for `Before21`. To see that `SearchMin` can reach $O(n \log n)$, consider `SearchMin` ($\langle 1, n+1\rangle, \{\langle 2, 2n+1\rangle, \langle 2, 2n\rangle, ..., \langle 2, n+2\rangle\}, \{\langle 2, 2\rangle, \langle 3, 3\rangle, ..., \langle n+1, n+1\rangle\}$).

`MarkBeforeMin12` is more complex, since it traverses C, Q and the same P (since the Q-node is searched back into P). All this contributes with $O(n \min(n, h \log d))$ (we have to add $O(n_P \min(n_P, h_P \log d_P))$ for each element of P, that searches into the same P). The two calls to `SearchMin` complete the analysis, to leave us also with $O(n_P(\min(n_C, h_C \log d_C) + \min(n_P \log d_C, h_P(d_P + d_C)) + \min(h_Q(d_Q + d_C), n_Q \log d_C))) = O(n \min(n \log d, hd))$.

We can use the same example as for `Before21` to show it can reach $O(n^2 \log n)$.

If P, Q and C are from the same view, the complexity changes exactly as it does in **before**$(k)(C)$.

The idea for **after**(C) is the same as for **before**(C), only some details differ. The complexity is also the same.

Merge version: For each element of P, we search for the nearest element in Q which is in the same minimal C-node and we mark it. We traverse P in depth-first order, so if a Q node is the candidate to mark a P node p, but it was also previously selected by another p', we must unmark the older (and farther) p'. To achieve this, the "mark" field of Q elements (called *marked* to ease reading) point to the (last and hence the only valid) P element they marked, if any. This way, the P node can be accessed and unmarked if it gets displaced by a newer one. Later, we collect the marked nodes of P.

P **before** Q (C) (*merge* version): Operate (P, Q)

```
Operate $(P,Q)$

Traverse $(P,Q)$.
Delete unmarked elements from $P$.

Traverse $(P,Q)$

While $To(p_i) < \infty$
        $p_i > q_j$ :  Pass $j$.
        $else$ :       $(c, lc) \leftarrow$ SearchC $(node(p_i))$.
                       $s \leftarrow \langle To(p_i)+1, \infty\rangle \cap c$.
                       $\{q_j ...\} \leftarrow$ Traverse $(query(p_i), \{q_j ...\})$.
                       While $(To(q_j) < From(s))$ Pass $j$.
                       $Qmarker \leftarrow$ Mark $(\{q_j ...\}, s, lc)$.$marker$.
                       If $Qmarker \neq none$
                          If $(marked(Qmarker) \neq none)$ Unmark $marked(Qmarker)$.
                          Mark $p_i$.   $marked(Qmarker) \leftarrow p_i$.
                       Pass i.
Return $\{q_j ...\}$.
```

Mark is which selects the minimum-distance candidate.

```
Mark $(\{q_j ...\}, s, \{c_i\})$

If $From(q_j) < From(s)$
   $(marker, \{c_i ...\}) \leftarrow$ Mark$(query(q_j), s, \{c_i ...\})$.
   If $(marker \neq none)$ Return $(marker, \lambda)$ else Pass $j$.

While $To(q_j) \le To(s)$
        $c_i < q_j$ :          Pass $i$.
        $c_i \supseteq q_j$ :  Pass $j$.
        $else$ :               Return $(q_j, \lambda)$.

If $(From(q_j) \le To(s))$ Return Mark$(query(q_j), s, \{c_i ...\})$.
Return $(none, \{c_i ...\})$.
```

This algorithm is analyzed as follows. The final deletion of unmarked nodes is $O(n_P)$. `SearchC` is $O(n_C h_C)$ in total as before. `Mark` is much like `Determ`, in the sense that it traverses a level of C in the whole invocation from `Traverse`, and of $O(d_Q)$ Q-nodes at each level of Q; and after the first time, just one of the two recursive calls can succeed, thus achieving $O(d_C + d_Q h_Q)$, or $O(d_C + n_Q)$ by observing that each element is worked on at most once. On the other hand, `Mark` is invoked once for each element of P, and the total number of `Pass` j performed is $O(d_Q)$ (the top-level of Q). Thus, we have $O(n_C h_C + d_Q + n_P(d_C + \min(n_Q, d_Q h_Q))) = O(n \min(n, dh))$ cost.

To see that it can reach $O(n^2)$, consider the same example of **before**(k).

If P, Q and C are from the same view, the complexity changes exactly as it does in **before**$(k)(C)$.

The idea for **after**(C) is the same as for **before**(C), only some details differ. The complexity is also the same.

5.2.2 Lazy Evaluation

The idea of lazy evaluation, which we mentioned briefly at the beginning of this chapter and in Section 5.1, is as follows. Instead of making an operation and obtaining the full tree, we represent that result as the operation to perform and the representation of the operands, that is $(X \ op \ Y)$, will be converted to a tree with root op and subtrees X and Y with no more computation. This technique would convert the evaluation of a query to just its syntax tree, except because the final result has to be obtained. To expand the final result, we begin by obtaining its top-level nodes. Suppose that we want to expand $(X \ op \ Y)$. We define algorithms to get the answer step by step, that is, to convert $(X \ op \ Y)$ to the top-level nodes $r_1..r_l$ with children $(X_1 \ op \ Y_1)..(X_l \ op \ Y_l)$. Of course that algorithm would need some information on X and Y, which is obtained by expanding the expressions that define them, node by node. Once we have the top-level nodes, we obtain their children, and then the children of each child, and so on, until obtaining the whole answer. Hopefully, not all the sets involved in the expression need to be fully evaluated (see for example [GT87]). This mechanism is not new, for example is of widespread use in lazy functional languages [Dav92].

5.2.2.1 Mechanism of Expansion

The type of our trees is now more complex, since they can be not expanded. For each view V, we have the type of *lazy trees* LT_V over V:

$$\begin{aligned}
LT_V &= NonEvaluated_V \cup Evaluated_V \\
NonEvaluated_V &= \bigcup_{op \in OP} op \times LT_V^{arity(op)} \\
Evaluated_V &= LSubtree_V^* \\
LSubtree_V &= N_V \times LT_V
\end{aligned}$$

where OP is the set of our operators, each one with its *arity*. Note that $Evaluated$ means evaluated at least one level. Note also that $T_V \subseteq LT_V$.

Suppose we have defined all our operations to obtain the first-level descendants of $(X \ op \ Y)$, and that we are given the syntax tree of a query. Then we present a procedure to obtain the expanded tree, expanding when necessary (and *only* when necessary) the subexpressions.

We use the following notation here:

- $(root_1(T_1)..root_k(T_k))$ is an $(Evaluated)$ lazy tree with those $root_i$ nodes comprising its first-level expansion, and T_i its descendants.
- $op(X_1, ..., X_k)$ is a $(NonEvaluated)$ lazy tree.
- $expand : LT_V \rightarrow T_V$ is our expansion function.
- $expandLevel : LT_V \rightarrow Evaluated_V$ expands, if it is not already expanded, one level of the lazy tree.
- $perform : OP \times LT_V^* \rightarrow Evaluated_V$, performs the operation on the arguments, to expand one level of the lazy tree.

Now, our procedure is defined as follows

$$\begin{aligned}
&expand((r_1(T_1)..r_k(T_n))) = (r_1(expand(T_1))..r_k(expand(T_k))) \\
&expand(op(X_1, .., X_k)) = expand(expandLevel(op(X_1, .., X_k))) \\
&expandLevel((r_1(T_1)..r_k(T_n))) = (r_1(T_1)..r_k(T_n)) \\
&expandLevel(op(X_1, .., X_k)) = perform(op, expandLevel(X_1), .., expandLevel(X_k))
\end{aligned}$$

Note that *expandLevel* is prepared to be invoked on an already evaluated tree. The reason for this is that the query tree is not necessarily a tree, it may well be a DAG. This way, an operation can be invoked on an operand which is already semievaluated by a previous operation, so that no expansion is made twice.

Observe that this idea is well-suited for interactive systems that combine querying with navigation (e.g. [Mac91, Mac90]). We can use *expandLevel* instead of *expand*, to obtain just the top-level list of the answer. This list can be presented to the user, who may want to discard parts of it and to enter into others. At this point we call *expandLevel* in the nodes the user wants to get in. This way, we can save significant work.

5.2.2.2 Implementation Considerations

Some considerations arise here. Observe that, since the flow of control of expansions is not easily predictable with lazy evaluation, it is hard to know when an element will no longer be used, since the operations carried out on it do not begin and end at predictable moments, but evolve as long as the values are needed. This implies that no deallocation can be done in advance, and thus that there is no benefit in modifying operations, since they are applicable if one knows that the operand is not referenced from elsewhere. The only situation in which one can assure that is when the operand is used only for one operation. The best option would be, therefore, to use modifying versions of operators in this last case and generating versions elsewhere. Under this schema, all remaining nodes can only be deallocated when the whole evaluation completes (but recall our comments about the need for a serious study of this problem).

Another problem is regarding swapping policy, if we keep some operands on disk. In lazy evaluation, it is too difficult to decide which nodes are good for swapping out, while in full evaluation it is easy to know which operands are to be needed later than which. This problem can be alleviated by the fact that lazy evaluation is likely to reclaim less space than full evaluation.

5.2.2.3 Description of the Algorithms

We now present the *lazy* version of our algorithms. Not that some choices we had for full evaluation are unavailable here, since we have to expand in a fixed order. We use in general a modification of the *merge* approach, but in some cases we lose efficiency in the process. Sometimes we must elect between redoing a little of work or expanding parts that perhaps will never be needed. We have systematically preferred to expand as little as possible.

"+" operator

The idea is the same as the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This makes the evaluation of the whole tree not so efficient, but this is compensated by the possibility of not evaluating the whole tree. Observe that we need additional information when we left the operation for later. We write that information as a subscript of the operator (it is initially $limit = \infty$).

Recall that no overlaps are possible, since P and Q must be from the same view. We present the algorithm in the next page.

This algorithm is not linear as its full version, because it is not longer valid that at each comparison at least one element from P or Q is solved and no longer compared. Observe that now, when $p_i \subset q_j$, we pass through the p_i's which will be the operands of the next level in order to continue the merge at the current level. Indeed, we may pass through them many times, each time the list descends one level. By observing each element of P, we see that it can be compared at most h times. By observing each element of Q, we see that it can traverse a level of P just once, that is $O(d)$. Thus, the algorithm is $O(n \min(d,h))$, i.e. $O(n^2)$ in the worst case. We cannot refine the analysis in terms of the sizes of P and Q, because of the switching in the case $p_i \supset q_j$.

To see that it can reach $O(n^2)$ consider $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\} \; + \; \{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$.

On the other hand, we work $O(d_P + d_Q) = O(d)$ per lazy invocation to this operator. Observe that it does not force any further expansion of the subtrees.

$P +_{limit} Q$ (*lazy* version): Operate $(P, Q, limit)$

```
Operate $(P, Q, limit)$

$res \leftarrow \lambda$.
While $\min(To(p_i), To(q_j)) < limit$
    $p_i < q_j$ :   $res \leftarrow res : p_i$.   Pass $i$.
    $p_i > q_j$ :   $res \leftarrow res : q_j$.   Pass $j$.
    $p_i = q_j$ :   $res \leftarrow res : (node(p_i), +_\infty(query(p_i), query(q_j)))$.   Pass $i, j$.
    $p_i \subset q_j$ :   $res \leftarrow res : (node(q_j), +_{To(q_j)+1}(\{p_i ...\}, query(q_j)))$.   Pass $j$.
                While $(p_i \subset q_j)$ Pass $i$.
    $p_i \supset q_j$ :   Exchange $P \leftrightarrow Q$.
Return $res$.
```

"−" operator

The idea is the same as for the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This makes the evaluation of the whole tree not so efficient, but this is compensated by the possibility of not evaluating the whole tree.

Recall that no overlaps are possible, since P and Q must be from the same view.

$P - Q$ (*lazy* version): Operate (P, Q, ∞).result

```
Operate $(P, Q, plimit)$

$res \leftarrow \lambda$.
While $To(p_i) < plimit$
    $p_i < q_j$ :   $res \leftarrow res : p_i$.   Pass $i$.
    $p_i > q_j$ :   Pass $j$.
    $p_i = q_j$ :   $p_i \leftarrow expandLevel(p_i)$.   $\{p_i ...\} \leftarrow query(p_i) : \{p_{i+1} ...\}$.
    $p_i \subset q_j$ :   $q_j \leftarrow expandLevel(q_j)$.   $Op \leftarrow$ Operate $(\{p_i ...\}, query(q_j), To(q_j) + 1)$.
                $res \leftarrow res : Op.result$.   $\{p_i ...\} \leftarrow Op.rest$.   Pass $j$.
    $p_i \supset q_j$ :   $res \leftarrow res : (node(p_i), -(query(p_i), \{q_j ...\}))$.   Pass $i$.
Return $(result = res, rest = \{p_i ...\})$.
```

This algorithm is not linear as its full version. The argument is almost the same as with "+", this time the case which produces the problem is $p_i \supset q_j$. The complexity is then $O(\min(n_P + n_Q h_P, n_Q + n_P d_Q)) = O(n \min(d, h))$.

To see that it can reach $O(n^2)$ consider $\{\langle 1, 2n-1\rangle, \langle 2n, 2n\rangle, \langle 1, 2n-2\rangle, \langle 2n-1, 2n-1\rangle, ..., \langle 1, n\rangle, \langle n+1, n+1\rangle\} - \{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$.

This operator can force the full trees to be expanded, for example if $P = Q$, thus it can work up to $O(n_P + n_Q) = O(n)$ in a single lazy invocation (as the full *merge* "−").

"is/same" operators

The idea for **is** is the same as for the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This does not introduce inefficiencies.

Recall that no overlaps are possible, since P and Q must be from the same view.

P **is** Q (*lazy* version): Operate (P, Q, ∞).result

```
Operate $(P, Q, plimit)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

    $p_i < q_j$ :  Pass $i$.
    $p_i > q_j$ :  Pass $j$.
    $p_i = q_j$ :  $res \leftarrow res : (node(p_i),$ is $(query(p_i), query(q_j)))$.   Pass $i, j$.
    $p_i \subset q_j$ :  $q_j \leftarrow expandLevel(q_j)$.   $Op \leftarrow$ Operate $(\{p_i ...\}, query(q_j), To(q_j) + 1)$.
               $res \leftarrow res : Op.result$.   $\{p_i ...\} \leftarrow Op.rest$.   Pass $j$.
    $else$ :       $p_i \leftarrow expandLevel(p_i)$.   $\{p_i ...\} \leftarrow query(p_i) : \{p_{i+1} ...\}$.

Return $(result = res, rest = \{p_i ...\})$.
```

This algorithm is linear as its full version, with the same argument: each comparison eliminates at least one element from the problem. The difference with "–" is that each time we leave something for later, it is a whole subtree, which we do not need to re-traverse.

To see that it can reach $O(n)$, we use the same example of the full version.

This operator can work $O(n_P + n_Q) = O(n)$ in a single lazy call, making the full trees to be expanded. Consider for example $\{\langle 1, n+1\rangle, \langle 1, n\rangle, ..., \langle 1, 2\rangle\}$ **is** $\{\langle 1, 2n+1\rangle, \langle 1, 2n\rangle, ..., \langle 1, n+2\rangle, \langle 1, 1\rangle\}$.

The **same** operation is exactly like **is**, except in that we do not check for node equality but for segment equality. In this case, overlaps are possible, so we use the same algorithm as for **is**, the *else* clause standing also for overlaps. The complexity is also the same.

"in" operator

The idea is the same as the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This does not introduce inefficiencies. The algorithm is exactly as in the full *merge* version, just inserting $p_i \leftarrow expandLevel(p_i)$ as the first instruction of the *else* clause.

Recall that overlaps are possible here, since P and Q can be from different views.

This algorithm is analyzed exactly as its full version, in fact it is identical except for the *expandLevel*. Note that it will never make Q expand more than the first level, although it can cause the full P to be expanded, thus it can be $O(\min(n_P + n_Q, d_P d_Q h_P)) = O(\min(n, d^2 h))$ in a single lazy call. Consider for example $\{\langle 1, n\rangle, \langle 1, n-1\rangle, ..., \langle 1, 1\rangle\}$ **in** $\{\langle 1, 1\rangle\}$.

The algorithm is not better if P and Q are from the same view.

"beginin/endin" operators

The idea is the same as for **in**. Recall that overlaps are possible here, since P and Q can be from different views.

P **beginin** Q (*lazy* version): Operate (P, Q)

```
Operate $(P,Q)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

    $p_i < q_j$ :           Pass $i$.
    $p_i > q_j$ :           Pass $j$.
    $p_i \subset q_j$ :           $res \leftarrow res : p_i$.   Pass $i$.
    $From(p_i) \in q_j$ :   $res \leftarrow res : (node(p_i),$ beginin $(query(p_i), \{q_j ...\}))$.   Pass $i$.
    $else$ :                $p_i \leftarrow query(expandLevel(p_i)) : \{p_{i+1} ...\}$.

Return $res$.
```

P **endin** Q (*lazy* version): Operate (P, Q)

```
Operate $(P,Q)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

    $p_i < q_j$ :   Pass $i$.
    $p_i > q_j$ :   Pass $j$.
    $p_i \subset q_j$ :   $res \leftarrow res : p_i$.   Pass $i$.
    $else$ :        $oldj \leftarrow j$.  While $(To(q_j) < To(p_i))$ Pass $j$.
                    If $(To(p_i) \in q_j)$  $res \leftarrow res : (node(p_i),$ endin $(query(p_i), \{q_{oldj} ...\}))$.
                    else
                        $p_i \leftarrow expandLevel(p_i)$.
                        $res \leftarrow res :$ Operate $(query(p_i), \{q_{oldj} ...\})$.
                    Pass $i$.

Return $res$.
```

These algorithms are not linear as their full versions. Observe that the top-level of Q can be retraversed many times, more exactly, for a complete path of P (then we do $O(h_P d_Q)$ work in Q). By adding the $O(n_P)$ traversal of P, we have $O(n_P + h_P d_Q)$. On the other hand, both algorithms are $O(d_P d_Q h_P)$ as their full versions, thus the final complexity is $O(\min(n_P + h_P d_Q, d_P d_Q h_P)) = O(\min(n + dh, d^2 h))$.

To see that they can reach $O(n^2)$, consider $\{\langle 1, 2n-1\rangle, \langle 2n, 2n\rangle, \langle 1, 2n-2\rangle, \langle 2n-1, 2n-1\rangle, ...\langle 1, n\rangle, \langle n+1, n+1\rangle\}$ **beginin** $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$, and $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n\rangle\}$ **endin** $\{\langle 1,1\rangle, \langle 2,2\rangle, ..., \langle n,n\rangle\}$.

These algorithms never force Q to be expanded, but they can expand the whole P. **beginin** can work $O(\min(n_P + n_Q, d_P d_Q h_P)) = O(\min(n, d^2 h))$ per call, consider for example $\{\langle 1, 2n+1\rangle, \langle 1, 2n\rangle, ..., \langle 1, n\rangle\}$ **beginin** $\{\langle 2,2\rangle, \langle 3,3\rangle, ..., \langle n+1, n+1\rangle\}$. **endin** can work $O(\min(n_P + h_P d_Q, d_P d_Q h_P)) = O(\min(n + dh, d^2 h))$ per call, the same example shows that it can reach $O(n^2)$.

If P and Q are from the same view, **beginin/endin** should be interpreted as **in**.

"[*s*]in/beginin/endin" operators

The idea for **in** is the same as the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This does introduce some inefficiencies. The important modifications are in that we need two marks for lazy deletion of unmarked nodes (the second mark means that the descendants of a node have some opportunity to be included).

Recall that overlaps are possible here, since P and Q need not to be from the same view.

$[s]$ P **in** Q (*lazy* version): Operate (P, Q)

```
Operate $(P,Q)$

Traverse $(P,Q)$.
$res \leftarrow \lambda$.

While $To(p_i) < \infty$
   If $p_i$ is marked
      If ($p_i$ is marked2) $res \leftarrow res : p_i$.
      else $res \leftarrow res : (node(p_i), \lambda)$.
   else
      If ($p_i$ is marked2) $res \leftarrow res : query(expandLevel(p_i))$.
   Pass $i$.

Return $res$.
```

```
Traverse $(P,Q)$

While $\max(To(p_i), To(q_j)) < \infty$

   $p_i < q_j$ :     Pass $i$.
   $p_i > q_j$ :     Pass $j$.
   $p_i \subset q_j$ :     $q_j \leftarrow expandLevel(q_j)$.   $pos \leftarrow 1$.
                Traverse $(\{p_i ...\}, query(q_j))$.
                While $p_i \subset q_j$
                   If ($p_i$ is unmarked and $pos \in s$) Mark $p_i$.
                   Pass $i$.   $pos \leftarrow pos + 1$.
   $else$ :       $query(p_i) \leftarrow [s]$in $(query(p_i), \{q_j ...\})$.
                Mark2 $p_i$.  Pass $i$.
```

This algorithm is analyzed almost as for its full version. In this case, it is also possible to repeat work on a level of Q, thus the total $= O(\min(n_P h_Q + n_Q, n_Q d_P + n_P) + \min(n_Q h_P + n_P, n_P d_Q + n_Q)) = O(n \min(h, d))$.

To see that it can reach $O(n^2)$ and that both P and Q can be fully expanded in a single call (thus being $O(\min(n_P h_Q + n_Q, n_Q d_P + n_P)) = O(n \min(h, d))$ in a single lazy call), consider $[s]$ $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$ **in** $\{\langle 1, 3n\rangle, \langle 1, 3n-1\rangle, ..., \langle 1, 2n+1\rangle\}$.

This algorithm is not better if P and Q are from the same view.

The $[s]$**beginin** and $[s]$**endin** operations are quite the same as $[s]$**in**, the only difference being that we should replace (in both places) the condition $p_i \subset q_j$ by $From(p_i) \in q_j$ for **beginin** and by $To(p_i) \in q_j$ for **endin**. The analysis is the same.

If P and Q are from the same view, $[s]$**beginin/endin** should be interpreted as $[s]$**in**.

"with/withbegin/withend(k)" operators

We begin by solving **with**. This operation is quite complicated to perform in a lazy fashion, mainly because we cannot know the size of the operands, since they are not computed yet. Moreover, we cannot even have the size computed for the expanded part of operands, since they are shared and a policy of immediate actualization of all pointers to a query is expensive and contrary to the spirit of lazy evaluation. Thus, we have a computed size, which is only guaranteed to be less than the real size. In order to expand the minimum possible of Q, we make up to three passes on the interesting part: counting the computed size at the top level, traversing the whole expanded part to actualize the computed size, and expanding if necessary. To avoid traversing a tree to expand any remaining node when we already know that it is fully expanded, we use an additional flag *compl*.

P **with**(k) Q (*lazy* version): Operate (P,Q).result

```
Operate $(P, Q, plimit)$

$res \leftarrow \lambda$.
While $To(p_i) < plimit$
        $p_i < q_j$ :  Pass $i$.
        $p_i > q_j$ :  Pass $j$.
        $p_i \subseteq q_j$ :  $q_j \leftarrow expandLevel(q_j)$.
                 $Op \leftarrow$ Operate $(\{p_i ...\}, query(q_j), To(q_j) + 1)$.
                 $res \leftarrow res : Op.result$.   $\{p_i ...\} \leftarrow Op.rest$.
        $else$ :       If Determ$(node(p_i), \{q_j ...\}, k)$
                     $res \leftarrow res : (node(p_i),$ with$(k)$ $(query(p_i), \{q_j ...\}))$.
                 Pass $i$.
Return $(result = res, rest = \{p_i ...\})$.
```

`Determ` determines if P contains k or more Q-nodes, making up to three passes if necessary. The list l can be made implicit, to avoid the space overhead.

`Actual` traverses Q, counting the nodes of the already expanded part, until no more nodes are found or until it finds k nodes included in p. It also actualizes the information of *size*, and returns a list of pointers where expansion can be made if necessary.

`Expand` expands Q, until there are no more nodes or it finds k nodes included in p.

```
Determ $(p, Q, k)$

pass 1:
   $t \leftarrow 0$.  $oldj \leftarrow j$.  If $(q_j$ $overlaps$ $p)$ Pass $j$.
   While $q_j \subset p$
         $k \leftarrow k - 1$.  $t \leftarrow t + size(q_j)$.
         If $(t \geq k)$ Return $true$.
         Pass $j$.
pass 2:
   $j \leftarrow oldj$.  $sz \leftarrow 0$.  $l \leftarrow \lambda$.
   While $q_j \not\supset p$
      $(t, nl) \leftarrow$ Actual$(query(q_j), k - sz, p, q_j \subseteq p)$
      $sz \leftarrow sz + t$.  $l \leftarrow l : nl$.  Pass $j$.
      If $(sz \geq k)$ Return $true$.
pass 3:
   $k \leftarrow k - sz$.  $sz \leftarrow 0$.
   For each $l_i \in l$
      $sz \leftarrow sz$ + Expand $(l_i, k - sz, p)$.
      If $(sz \geq k)$ Return $true$.
Return $false$.
```

```
Actual $(Q, k, p, incl)$

If $Q$ is expanded
   If $(incl \ \wedge \ (Q.compl \vee size(Q) \geq k))$ Return $(size(Q), \lambda)$.
   $sz \leftarrow 0$.   $l \leftarrow \lambda$.
   While $(q_j < p)$ Pass $j$.
   While $(q_j \not> p \ \wedge \ sz < k)$
      $(t, nl) \leftarrow$ Actual $(query(q_j), k - sz - 1, p, q_j \subset p)$.
      $sz \leftarrow sz + t$.   $l \leftarrow l : nl$.
      If $(q_j \subset p)$ $sz \leftarrow sz + 1$.
   if $(size(Q) < sz)$ $size(Q) \leftarrow sz$.
   If (the While traversed the whole $query(Q) \ \wedge \ incl \ \wedge \ l = \lambda$) $Q.compl \leftarrow true$.
   If $(sz \geq k)$ $l \leftarrow \lambda$.
   Return $(sz, l)$.
else Return $(0, \{Q\})$.
```

```
Expand $(Q, k, p)$

$Q \leftarrow expandLevel(Q)$.   $t \leftarrow 0$.
While $q_j \not> p \ \wedge \ t < k$

   $\begin{cases} q_j < p: & \text{Pass } j. \\ q_j \subset p: & t \leftarrow t + 1 + \text{Expand } (q_j, k - t - 1, p). \quad \text{Pass } j. \\ else: & t \leftarrow t + \text{Expand } (q_j, k - t, p). \quad \text{Pass } j. \end{cases}$

Return $t$.
```

`Expand` expands a node, completely or until it finds k nodes. `Actual` traverses the tree, searching for already expanded nodes, completely or until it finds k nodes, and actualizes the computed sizes if appropriate. `Determ` traverses Q, stopping when it finds k nodes or more.

It may be argued that `Expand` can cause expansion of an unbounded number of nodes that result in empty queries that do not reduce k. But observe that for each unexpanded query included in the list that `Expand` processes, there is an expanded parent which has been counted. If the counted nodes before calling to `Expand` did not reach k, then the length of the list is $< k + 2$. We add 2 for the possibility of overlapping nodes that add an element to the list without being counted. Therefore, `Expand` cannot be invoked more than $O(k)$ times from outside.

If at each recursive call, `Expand` reduced k by one, then the total work done by `Expand` would be $O(k)$, but it can also expand nodes that overlap with p, what does not reduce k. This can be done for a complete path of Q, and makes it traverse the levels. This way, `Expand` is $O(k + d_Q h_Q)$ for the whole set of invocations for a single p node. Since it cannot traverse more than the whole Q, it is in fact $O(\min(n_Q, k + d_Q h_Q))$.

`Actual` is quite similar: cannot work more than $O(k)$ time, except for overlaps, what makes it $O(\min(n_Q, k + d_Q h_Q))$ for the whole set of invocations for a single p node.

This way, `Determ` is also $O(\min(n_Q, k + d_Q h_Q))$, and the whole algorithm is $O(n_P \min(n_Q, k + d_Q h_Q)) = O(n \min(n, k + dh))$.

To show that it can reach $O(n^2)$, we present the following example: $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$ **with**(n) $\{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$.

This operation cannot cause further expansion in P, since if a P node is not included, its subtree is neither included. But it can expand Q wholly. At each lazy call, the algorithm can work $O(\min(n, d(k + dh)))$. To see this, consider that, on the one hand, we work only on the top-level of P, thus the argument we used for the full expansion shows that (replacing n_P by d_P) we can work at most $O(d_P(k + d_Q h_Q))$ (not considering the n_Q bound yet). On the other hand, consider that in the whole execution of the lazy call, `Determ` cannot touch a Q node more than twice (since no more than two top-level P nodes can contain or overlap the Q node), thus we can take a better minimum: $O(\min(n_Q + d_P, d_P(k + d_Q h_Q))) = O(\min(n, d(k + dh)))$.

To see that it can reach $O(n)$, consider the same example as for "+".

If P and Q are from the same view, this algorithm performs better. The reason is that now **Expand**, **Actual** and **Determ** are really $O(\min(n_Q, k)$, thus the whole algorithm is $O(n_P \min(n_Q, k)) = O(n \min(n, k))$. The maximum amount of work done at each call is also smaller, since we can eliminate the $d_Q h_Q$ added because of overlaps, to reach $O(\min(n_Q + d_P, d_P k)) = O(\min(n, kd))$.

The same examples of the general case can be used to show that even when P and Q are from the same view, the whole complexity can reach $O(n^2)$ and the complexity per call can reach $O(n)$.

The operations **withbegin/withend**(k) are similar to **with**(k), we need minimal changes to the algorithm. These are: in both `Actual` and `Expand`, the condition $q_j \subset p_i$ should change to $From/To(q_j) \in p_i$; and in the "`pass 1`" of `Determ` we should, for **withbegin**, add an instruction after the `While` block, namely "`If` $(q_j$ *overlaps* $p)$ $k \leftarrow k - 1$. `If` $(t \geq k)$ `Return` *true*"; and for **withend**, do also "$k \leftarrow k - 1$" if the first `If` holds. The analysis is the same.

If P and Q are from the same view, **withbegin/withend**(k) should be interpreted as **with**(k).

"[s]child"

The idea is the same as the *merge* version of full evaluation, the difference being that the subproblems are left to be solved later. This does not introduce inefficiencies. Recall that initially $par = none$, and that overlaps are not possible here, since P and Q must be from the same view.

[s] P **child**$_{par}$ Q (*lazy* version): Operate (P, Q, par).result

```
Operate $(P, Q, par)$

$res \leftarrow \lambda$.
While $To(p_i) < (par = none\ ?\ \infty\ :\ To(par) + 1)$

    $p_i < q_j$ :   If $par \neq none \wedge parent_{view}(p_i) = par \wedge pos_{view}(p_i) \in s$
                        $res \leftarrow res : (node(p_i), \lambda)$.
                    Pass $i$.
    $p_i > q_j$ :   Pass $j$.
    $p_i \supseteq q_j$ :   If $par \neq none \wedge parent_{view}(p_i) = par \wedge pos_{view}(p_i) \in s$
                        $res \leftarrow res : (node(p_i), [s]\mathbf{child}_{par}(query(p_i), \{q_j ...\}))$.
                        While $(p_i \supseteq q_j)$ Pass $j$.
                    else
                        $p_i \leftarrow expandLevel(p_i)$.
                        $Op \leftarrow$ Operate $(query(p_i), \{q_j ...\}, par)$.
                        $res \leftarrow res : Op.result$.   $\{q_j ...\} \leftarrow Op.restq$.
                    Pass $i$.
    $p_i \subset q_j$ :   $q_j \leftarrow expandLevel(q_j)$.
                    $Op \leftarrow$ Operate $(\{p_i ...\}, query(q_j), node(q_j))$.
                    $res \leftarrow res : Op.result$.   $\{p_i ...\} \leftarrow Op.restp$.   Pass $j$.

Return $(result = res, restp = \{p_i ...\}, restq = \{q_j ...\})$.
```

This algorithm is $O(n_P + n_Q) = O(n)$ as its full version, despite it looks like other $O(n^2)$ operations. The possible problem here is the case $p_i \supseteq q_j$, since it traverses some q_j's again after having processed them. But the difference is that the list of Q-children of a Q-node is re-traversed only once in the whole process, namely when we find the view children of Q in P (since all children of a node are in a single level in any possible query tree), thus it adds $O(n)$ to the whole process.

To see that it can reach $O(n)$ consider the same example of the full version.

It is also possible to work $O(n_P + n_Q) = O(n)$ at a single lazy call, and to expand fully P and Q, consider for example $[s]\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle, \langle 1, 1\rangle\}$ **child** $\{\langle 1, n\rangle, \langle 1, n-1\rangle, ..., \langle 1, 2\rangle\}$, when no parent/child relationship holds.

"parent(k)" operator

The idea is the same as for the *merge* version of full evaluation, but leaving subproblems for later. Since this time we have to traverse the children of a P-node to determine which of them are children in the view (to know if it remains or not), we introduce an inefficiency, since the children have to be traversed later for the descendants of the P-node.

Recall that overlaps are not possible here, since P and Q must be from the same view.

P **parent**(k) Q (*lazy* version): Operate (P, Q, ∞).result

```
Operate ($P, Q, plimit$)

$res \leftarrow \lambda$.
While $To(p_i) < plimit$

    $p_i < q_j$ :     Pass $i$.
    $p_i > q_j$ :     Pass $j$.
    $p_i \supset q_j$ :     $children \leftarrow 0$.   $oldj \leftarrow j$.
                  While $p_i \supset q_j$
                      If ($parent_{view}(q_j) = p_i$) $children \leftarrow children + 1$.
                      Pass $j$.
                  If $children < k$
                      $p_i \leftarrow expandLevel(p_i)$.
                      $res \leftarrow res$ : Operate ($query(p_i), \{q_{oldj}...\}, \infty$).$result$.
                  else $res \leftarrow res : (node(p_i), \mathbf{parent}(k)\ (query(p_i), \{q_{oldj}...\}))$.
                  Pass $i$.
    $p_i \subseteq q_j$ :     $q_j \leftarrow expandLevel(q_j)$.
                  $Op \leftarrow$ Operate ($\{p_i...\}, query(q_j), To(q_j) + 1$).
                  $res \leftarrow res : Op.result$.   $\{p_i...\} \leftarrow Op.rest$.   Pass $j$.

Return ($result = res, rest = \{p_i...\}$).
```

This algorithm is not linear as its full version, since it can pass many times over the same Q-nodes. Observe that for each P-node one can work at most $O(d_Q)$, and that on the other hand, one cannot repeat the work on a single Q-node more than $O(h_P)$ times. This way, we obtain $O(\min(n_Q + n_P d_Q, n_P + n_Q h_P)) = O(n \min(d, h))$.

To see that it can reach $O(n^2)$, consider $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$ **parent**(1) $\{\langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$.

This operation can force P and Q to be fully expanded, and can work $O(\min(n_Q + n_P d_Q, n_P + n_Q h_P)) = O(n \min(d, h))$ at a single lazy call. To see this, consider $\{\langle 1, 2n\rangle, \langle 1, 2n-1\rangle, ..., \langle 1, n+1\rangle\}$ **parent**(1) $\{\langle 1, 3n\rangle, \langle 1, 3n-1\rangle, ..., \langle 1, 2n+1\rangle, \langle 1, 1\rangle, \langle 2, 2\rangle, ..., \langle n, n\rangle\}$, when no parent/child relationship holds.

"before/after(k)(C)" operator

We begin by **before**. The idea is exactly as in the full *merge* version, but leaving subproblems for later. This force us to retraverse some Q nodes that in the full version are advanced by the children of the node, and this makes marking not worth anymore. The initial value for *qlimit* is ∞. Recall that overlaps are possible here, since P, Q and C can be from different views.

P **before**$_{qlimit}(k)$ Q (C) (*lazy* version): Operate $(P, Q, qlimit)$

```
Operate $(P, Q, qlimit)$

$res \leftarrow \lambda$.
While $To(p_i) < \infty$

  { $p_i > q_j$ :  Pass $j$.
  { $else$ :       $(c, lc) \leftarrow$SearchC $(node(p_i))$.
  {                $lim \leftarrow \min(qlimit, To(c))$.
  {                $s \leftarrow \langle To(p_i)+1, To(p_i)+k\rangle \cap \langle 1, lim\rangle$.
  {                $oldj \leftarrow j$.  While $(To(q_j) < From(s))$ Pass $j$.
  {                If (Determ $(\{q_j ...\}, s, lim, lc)$.$tomark$)
  {                    $res \leftarrow res : (node(p_i),$ $\mathbf{before}_{lim}(k)$ $(query(p_i), \{q_{oldj}...\})(lc)$.  Pass $i$.
  {                else $\{p_i ...\} \leftarrow query(expandLevel(p_i)) : \{p_{i+1}...\}$.

Return $res$.
```

SearchC and Determ are exactly the same as their full *merge* versions, just taking care of invoking *expandLevel* on a node prior to using its *query*.

This algorithm is analyzed almost as its full version. The only change is that the total number of times "Pass j" is executed can be now $O(h_P d_Q)$, not $O(d_Q)$ as before, but this does not change the total.

We can, in a single lazy call, work $O(n_C h_C + n_P(d_C + \min(n_Q, h_Q d_Q))) = O(n \min(n, dh))$. To see this, consider the same example as for the *merge* version.

It can also fully expand P, Q and C. To see this, consider $\{\langle 1,n\rangle, \langle 1,n-1\rangle, ..., \langle 1,1\rangle\}$ **before** $\{\langle 1,2n\rangle, \langle 1,2n-1\rangle, ..., \langle 1,n+1\rangle\}$ $(\{\langle 1,3n\rangle, \langle 1,3n-1\rangle, ..., \langle 1,2n+1\rangle\})$.

The change experimented by the complexity when P, Q and C are from the same view is exactly as for the *merge* version. The examples presented for the general case show that the maximum amount of work and expansion per lazy call does not change either, since the example can occur in a single hierarchy.

The idea for the **after**$(k)(C)$ operator is similar to **before**$(k)(C)$, except in some details. The complexity is also the same.

"before/after(C)" operators

We have not been able to develop a real *lazy* version for these operations, since the obtention of the first level requires a global computation involving the whole tree, and what happens with a node depends on its siblings and their descendants, so there is no foreseeable way to separate subproblems for later expansion. Anyway, although we have to obtain the whole answer, it is possible to delete nodes without expanding them, which is the purpose of lazy evaluation. The algorithm is, thus, the same, taking care of invoking *expandLevel* on a node prior to using its *query*. The complexity is of course the same as its full *merge* version.

5.2.3 Analysis Summary

In this section we summarize the space and time complexity results discussed in the presentation of algorithms.

5.2.4 Space for Queries

Since the representation of a set of Q-nodes (i.e. the answer to a query or subquery) is dynamic, it uses, for each node, a pointer to the view ($\log_2 |N_V|$ bits), a pointer to its children ($\log_2 |Q|$ bits, if they are stored contiguously), the number of children ($\log_2 |Q|$) and the size of the subtree ($\log_2 |Q|$). Since some algorithms rely on marks, we really need more information. The space requirements for marks range from 1 to $\log_2 |N_V|$ bits. Thus, ignoring marks, we need

$$S_Q = |Q|(\log_2 |N_V| + 3\log_2 |Q|) \leq 4|N_V|\log_2 |N_V|$$

bits per query result. If we consider $|Q| \approx |N_V|/|C_V|$ (which is reasonable), we need

$$S_Q \approx \frac{|N_V|}{|C_V|}\left(4\log_2 |N_V| - 3\log_2 |C_V|\right)$$

bits. This formula is the same if we use a linked list for the level of the tree, since in that case we replace the number-of-children field by a next-sibling field (both are $\log_2 |Q|$).

Measured in words, $S_Q = 4|Q|$. If we include the mark field, we have $S_Q = 5|Q|$.

During the resolution of a query, we need to keep many results at the same time. We saw in section 5.2.1.2 how to minimize space utilization if we use a tree for the query syntax (if we use a DAG it is much harder to predict the space utilization). If we have a query with a syntax tree of q nodes, the worst that can happen is that the tree is balanced. Suppose all operators are of arity a ($a \geq 2$, since unary operators do not force us to store more nodes), then by following the principle of solving the heaviest node first, we will have in the worst case $a - 1$ computed answers per level, thus needing a total space less than

$$S_Q\left(1 + (a-1)\log_a(1 + q(a-1))\right) = O\left(S_Q a \log_a q\right)$$

Note that in most cases $a = 2$. In that case, the exact answer is

$$S_Q\left(1 + \log_2(1 + q)\right)) = O\left(S_Q \log q\right)$$

In lazy evaluation we cannot predict how much space we will need, since we have normally all internal nodes partially computed.

If we use a DAG we may have to keep also space for all shared nodes.

5.2.5 Running Time of Operations

Since we have already analyzed the operations when we described them, we summarize and comment here the results.

As we can see in Table 5.1, the *merge* approach is better in general if we look at the worst case behavior. However, we must take into account other aspects of the problem. For example, the *search-operate* approach could be very good if the two operands are very distinct in size, we test this in the simulations. The *lazy* approach can also be better than the full one, because although it is less efficient in producing the whole trees, it can deliver the answer generating the intermediate solutions only to a partial degree.

We have not included here the operators to enhance the pattern-matching sublanguage, since they are quite easy to implement and normally linear. We have not included also the implementation of the **View** and **Constr** queries, since they are discussed in the section about indexing, and their performance depends on the index. Anyway, most implementations of **Constr** and **View** are linear in the size of the result.

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, coupling nodes with segments, using a data structure in which we can easily separate ranges of segments, and a language which reduces all queries to operations on proximal nodes lead us to an implementation where the amortized cost per retrieved element is, in many cases, constant.

In order to compare the efficiency of our approach to that of the other considered models, we must define how we measure the efficiency of those models.

The three simpler models (the hybrid model [BY94], PAT expressions [ST92] and overlapped lists [CCB95a]) have to manage sequential lists (no nesting) in their intermediate results, thus they are susceptible of an

Operation	**Search-Operate**	**Merge**	**Lazy** (in whole)	**Lazy** (per call)
$+$	$n \log d .. n \log^2 d$	n	$n \min(d, h)$	n
$-$	$n \log d .. n \log^2 d$	n	$n \min(d, h)$	n
is	$n \log d .. n \log^2 d$	n	n	n
same	$n \min(n, h \log d)$ $[n \log d .. n \log^2 d]$	n	n	n
in	$n \log d$	$\min(n, d^2 h)$	$\min(n, d^2 h)$	$\min(n, d^2 h)$
beginin	$n \log d$	$\min(n, d^2 h)$	$\min(n + dh, d^2 h)$	$\min(n, d^2 h)$
endin	$n \log d$	$\min(n, d^2 h)$	$\min(n + dh, d^2 h)$	$\min(n + dh, d^2 h)$
$[s]$***in**	$n \min(n, d + h \log d)$ $[nd]$	$n \min(d, h)$	$n \min(d, h)$	$n \min(d, h)$
with*(k)	$n \min(n, h \log d)$ $[n \log d .. n \log^2 d]$	n	$n \min(n, k + dh)$ $[n \min(n, k)]$	$\min(n, d(k + dh))$ $[\min(n, kd)]$
$[s]$**child**	$n \log d .. n \log^2 d$	n	n	n
parent(k)	$n \log d .. n \log^2 d$	n	$n \min(d, h)$	$n \min(d, h)$
after/before$(k)(C)$	$nh \log d$	$n \min(n, dh)$	$n \min(n, dh)$	$n \min(n, dh)$
after/before(C)	$n \min(n \log d, dh)$	$n \min(n, dh)$	$n \min(n, dh)$	$n \min(n, dh)$

Table 5.1: Complexity results for all versions of the operations. When the complexity can be better if the operands are in a single view, the better complexity is indicated below the general value, in square brackets.

implementation in which all operators are linear (a *merge*-like approach). They can also be implemented in a *search-operate*-like manner, i.e. searching each segment into the other set. This approach makes most operations $O(n \log n)$, but as *search-operate*, could be preferred in certain cases (see, however, the experimental results). Since the actual implementations are not a property of the models and not always are available (some of them are commercial products), we decide to consider these models as if they had a linear implementation (i.e. the best) for all operands, i.e. slightly better than ours.

The lists-of-references model is quite complex and hence has problems about efficiency [Mac91, Mac90]. However, since we are taking the part of the model similar to ours (i.e. discarding attributes and hypertext management), we only take into account the implementation of this subset. From the cited papers we can infer that the implementation of that subset makes most operations $O(n \log n)$, so we use that complexity.

The more complex models, instead, cannot have that good implementation. The tree matching model [KM93] involves a number of NP-Complete problems, especially regarding logical variables and unordered inclusion. Only ordered tree inclusion without variables can be similar in efficiency to our approach, but this is a too restricted version of the model. On the other hand, there is no data about the efficiency of *p-strings*, but many operations must be $O(n^2)$ (because of the cardinality of the result set). Moreover, there are some data about the times required to process the SOED that indicate that the implementation is not very efficient. Thus, we treat these complex models as being much less efficient than ours.

Chapter 6

A Prototype

In this chapter we describe a prototype that implements our model, following the system architecture proposed earlier. We have used this prototype to measure average time and space used, and to test different heuristics.

6.1 Implementation

Our prototype provides two main facilities

Indexing: each document is indexed in two separate forms: an index for text and another for structure. The index for text is provided by the text search engine we use. The index for structure can be built by any mechanism; for example we have implemented parsers for some markup languages and a tool to convert parse trees to the index format.

Querying: a search module implementing all representative operations is provided. This module has three implementations, one for each version of the algorithms. Pattern-matching subqueries are redirected to the text search engine that we use.

See Figures 6.1 and 6.2 for diagrams of the software architecture of our prototype. Observe how the general architecture described in Section 3.4 is mapped into this one. Currently, the interface layer is very primitive, and the document layer does not exist. There is no query parser yet, we manually write the query plans.

Currently, our text search engine is the API of SearchCity [Ars92], which is based on the use of PAT arrays [FBY92, chapter 5] (also called suffix arrays [MM90]), and partially implements the hybrid model [BY94]. PAT arrays allow to find all occurrences of a given prefix in $\log_2 n$ time, where n is the number of index points of the text. Its space requirement is n words. The matching sublanguage supported by the API includes: whole words, ranges, wildcards in any place but the beginning, proximity search, boolean operators, etc. It also has tools to define fields (much like [BY94]), but we do not use them. It provides a number of document management facilities, so the "document layer" of our architecture could in fact be implemented with the same API.

The API also allows to filter the raw text, by applying a character normalization filter, a synonyms and stopwords filter, and a format filter. This last filter allows to use files in other formats different from ASCII (e.g. Word), 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. The API provides a tool to add new format filters.

We use that filtering tool to incorpore 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 view index. This way, we have an index for matches and a separate index per view. The PAT array can be seen as a generic index for the text view. Recall that we do not want to allow users to see the markup, since all what they should do with it should be done through the operators to query on structure.

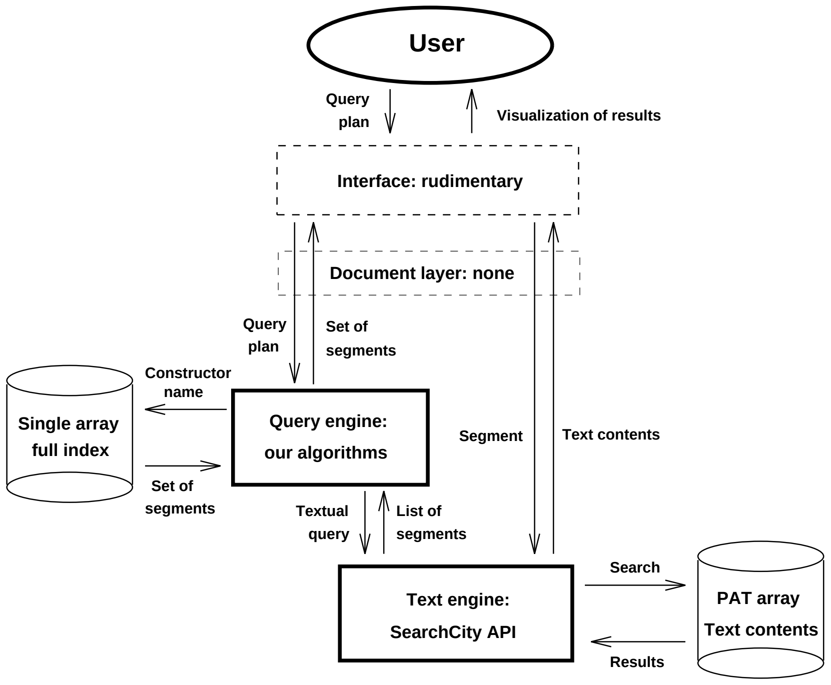


Figure 6.1: The architecture of our prototype regarding querying. Dashed components indicate that their implementation is still missing or rudimentary.

Thus, for each document format we add, we need to build a filter for the API, allowing it to discard the non-searchabe portion of the document (i.e. markup); and a parser to build the tree precisely from that discarded markup. That tree is converted to the index format we use for structure (i.e. the view array), by another tool that we implemented which eases this step. That tool isolates the problem of obtaining the structure from the problem of generating the index.

For some formats, we have not been able to truly filter the native files, being thus forced to create a new, filtered, copy of the originally marked file in the process of parsing. The reason for this is that the filters of the API of SearchCity are not powerful enough in some cases: a filter must be able to be set at any character of the file and to filter the text from there on in an automata-like fashion; and some markup systems would need an unbounded amount of memory per state (a stack, in fact) in order to do this.

At this moment, we have implemented simplified filters for some markup systems. They are simplified in the sense that they recognize some components of the markup, not all of it. The markup systems incorporated until now follow.

SGML [ISO86]: Stands for Standard Generalized Markup Language. It is the ISO standard on markup, so the ability to incorpore documents marked this way should give our system access to an important set of documents, especially if SGML becomes commonplace. We can recognize the structure of any markup system compliant with SGML. Since SGML allows complicated markup systems, in which things can be mentioned without really appearing in the document (by means of entities, etc.), we prefer to use a public parser, *sgmls V1.1*, to solve all those complications and to work on an easily interpretable version of the original file. We then parse that output file, obtaining a final file with only contents (to be indexed and queried as text) and a view file, with the corresponding structure. We do not analyze the semantics of SGML documents in depth, but only recognize structuring, subdocuments and commands, taking all them simply as constructors. Since the file we work on is not the original but a filtered version, there is no point in keeping it as a native file, thus we make a filtered copy from it.

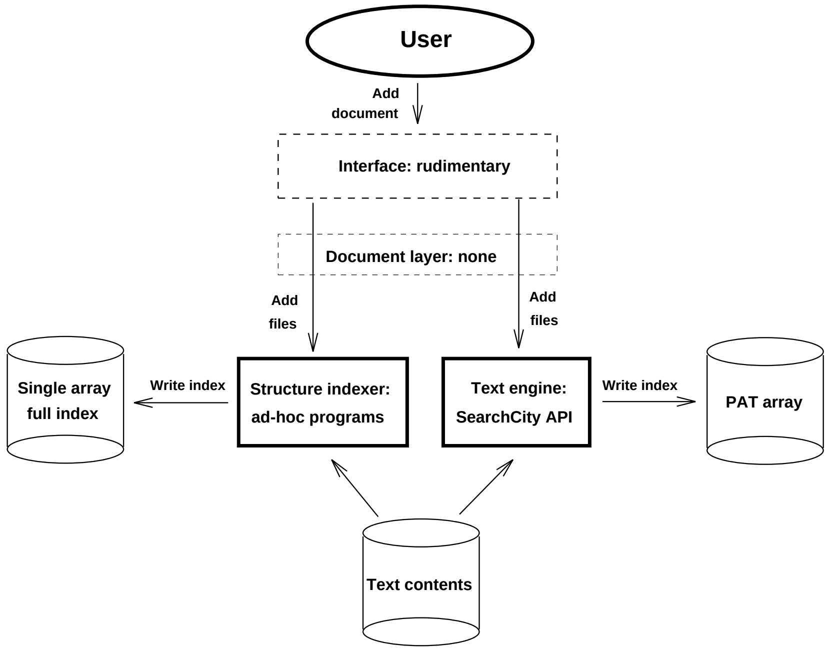


Figure 6.2: The architecture of our prototype regarding indexing. Dashed components indicate that their implementation is still missing or rudimentary.

LaTeX[Lam86]: Is a markup system oriented to define the structural and presentation aspects of books and articles, commonplace in the academic world. Recognizing LaTeX documents allows us to test the kind of structures likely to be found in books, for example. We recognize only the markup related to structuring the document. The rest of the commands, related to style and symbols, is ignored and not filtered. An interesting enhancement is to add a separate view for style commands and eliminating them from the text, while symbols are probably best dealed with by leaving them as they appear. Our parser assumes that the file has no structuring errors. Due to the complexity of the parsing process and to the exposed limitations of the filtering mechanism provided by the API, we build a new, filtered copy of the file, instead of making a filter for the native LaTeX file. A more space-efficient solution is to have a file of bits, saying which characters of the original file should be seen. No automata-based mechanism (such as the one provided by the API) with fixed amount of memory per state can deal with the problem of accessing an arbitrary character of the file and telling whether it should be included or not.

C code [KR78]: In this case, there is no markup (then no filtering is necessary), but the structure must be extracted by parsing the C language. Including C language structuring allows us to test our prototype on much finer-grain structures, likely to be found in most programming languages. We adapted a parser from a software visualization system developed at this department [BYJQ92], to obtain a parse tree augmented with segment information. The only simplification made here is that the files must be preprocessed (i.e. macros expanded, etc.), because the parser does not handle preprocessor directives (they cannot be simply ignored, since type information is necessary to succesfully parse a file, and it can be in other `#include`d file). A more sophisticated version should use the preprocessor file and line information to allow using the original file instead of the preprocessed one. But this feature demands multiple files management, an option that our prototype lacks. Since SearchCity allows multiple files, adding such a feature is a matter of better integration.

DDIF [Dig91]: Stands for Digital Document Interchange Format. It is a markup system designed by DEC and intended, among other things, to structure documents. The DDIF structure is normally quite fine-grain. We do not parse DDIF directly, but use the output of *ddis2ascii V3.03*, a program that outputs a readable version of a DDIF-marked document. We recognize the structuring commands and their scope, although we do not make a more complex semantic analysis, which would be required to capture the full power of the language. We filter the native file, not creating a copy.

We implemented the simplest version of the structure index, i.e. the full index in memory, together with procedures to load it from and save it to disk. A real application should use some of the solutions we propose for keeping the index on disk.

The operators we have implemented cover all interesting problems and algorithms: **View**, **Constr**, **Match**, $+$, $-$, **is**, **same**, **in**, $[s]$**in**, **with**(k), $[s]$**child**, **parent**(k), **before**$(k)(C)$, and **before**(C).

Text views are implemented by creating, for each match on queries, an index array, where one entry per node of the resulting tree is created. This is not the smartest way to do it, but it is simple enough for a prototype. This text view is deallocated when the query disappears.

The language for sequences (recall $[s]$**in**, $[s]$**child**, etc.) is as follows: one can express a finite number of ranges from three classes: from position i to j, from position $last - i$ to $last - j$, and from position i to $last - j$. We believe it is expressive enough for most normal cases. That means, for example, that we cannot select all even chapters, or all prime-numbered sections.

For the *search-operate* version, we implement the set of children of a node as a sorted array, in which we can easily perform binary or interpolation search. The problem is that many operations, if we consider the necessary merge of arrays or deletion of nodes, become $O(n^2)$ in the worst case. A real implementation should use a data structure as described earlier, to make all those operations in $O(\log d)$ time. We have not implemented also the binary split method to create artificial nodes, which assures that the worst case cannot reach $O(n^2)$. We instead search for each node of the level in sequence. Although a real application should implement binary split, the difference in the average case is negligible except for very deep trees (which are unlikely to appear in practice), while we can benefit in speed due to simpler code.

These two simplifications have been tested and found reasonable. On the one hand, we show in the tests that the behavior of *search-operate* is no different as a proportion of *merge* for "$-$" and **is** operations, in which "$-$" does only search but **is** does a lot of node deletion. On the other hand, we tested in the same examples of "$-$" (which do only searches) the strategy of artificial nodes vs the simpler code, and found them equivalent.

All *search-operate* versions destroy one or both operands (i.e. they are implemented as modifying operations).

For the *merge* and *lazy* versions, instead, we use sorted linked lists, which are efficient for merging operations (we do not search for elements here, we just traverse sets). *Merge* operators are also implemented as modifying operations.

Lazy evaluation is efficiently implemented by storing, as the identifier of the operator to apply, the address of its procedure (all procedures are normalized to receive the same type of parameters). The procedures used in the full-evaluate versions solve the queries and obtain the answers, while in lazy evaluation they just build the tree that indicates the operations to perform. Another procedure is invoked later to force expansion of the result.

A problem related to lazy evaluation is the need of a garbage collection mechanism for shared nodes. Since there are no circular references, either reference count or mark&sweep mechanisms can be used [Coh81]. While the first one deallocates immediately unused memory, the second one takes place only upon memory exhaustion and can be slower, stopping the execution for a long time and making evaluation time more unpredictable. On the other hand, mark&sweep can easily collect complex structures, including cycles.

Since we share lists of queries in a complex way (i.e. it is possible to point to the middle of a list), we prefer the mark&sweep style. We implement a pool of shared objects to deallocate all objects after the whole evaluation of a query. A real application should either adapt the reference count mechanism to deal with our lists, or use the pool to implement a mark&sweep garbage collection scheme, that upon memory exhaustion marks all what is reachable from the query that initially triggered the expansion, and then frees all what is in the pool and is not marked.

Lazy operations are not implemented as modifying but as generating operations (i.e. they do not destroy their operands). This is the only feasible alternative for shared objects. We can share queries and lists of queries independently, and we can even share parts of lists.

6.2 Experiments

In this section we present the results obtained with the prototype. We first present a list of questions that these tests help to answer, then we expose the tests we made and their results, and finally we extract some conclusions from experimental data.

6.2.1 Motivation

The list of questions the tests should help us to answer is:

- Which are the average times of the operators?
- The *search-operate* versions have normally two alternatives: search the first operand into the second or vice versa ("12" or "21"). Under which condition should each alternative be selected?
- Under which condition should one use *merge* or *search-operate* versions? We must compare running times to determine a global heuristic that selects to *merge* or to *search-operate* depending on the size characteristics of the operands (in this case, we should unify the data structures, probably by using a binary tree to arrange the nodes of each level).
- How much do we save by using lazy evaluation in terms of time? And in terms of space? Is it worth paying the overhead?
- Is it advisable to use interpolation instead of binary search in *search-operate* algorithms?
- How does the shape, depth and width of the structure affect the previous measures?

6.2.2 Tests

All tests were run 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.

We use three main document formats in our texts: LaTeX documents, which provide us with a sample representative of the distribution likely to be found in books and texts: relatively coarse-grain, shallow and wide; and C code, SGML and DDIF, which are much finer-grain, deeper and not so wide. On each document type we have only one hierarchy, thus we had no overlap cases to include in our tests. We provided these cases by means of text operations, that generate segments overlapping with structural components.

As explained in the initial exposition of the model, there is no good probabilistic modellization of the distribution of hierarchies, what prevents us to run a sound test and obtain results about mean, variance and confidence intervals. We instead selected a number of appropriate examples, to have an idea of the measures that we want to obtain. As it is shown later, this is enough for our purposes.

We conducted three main tests:

An operator-wise test: It is the most extense. Its aim is to obtain average values for all versions of all representative operators, answering then the questions about average times, and about which algorithms are better in each case. Details follow:

- The operators tested were: "+", "−", **is**, **in**, [s] **in**, **with**(k), [s] **child**, **parent**(k), **before**(k) and **before**.

- For each operator, we tested the five versions of the algorithms: *lazy*, full *merge* and three versions of *search-operate*: using the heuristic to select between "12" and "21" versions, using always "12" and using always "21".
- We ran the algorithms for nine different sizes of operands, i.e. for all combinations of sizes 10^2, 10^3 and 10^4 for both operands. The time for smaller sets was negligible. We used such a gross scale because of what we mentioned about the difficulty to run more precise tests.
- Each run was made on flat and deep operands, to see how the structure influenced the results.
- Finally, for each operation, algorithm version, size combination and flat/deep selection, five different operand pairs were selected, intended to represent a range of possible values on that case.
- Some special cases: in both **before** operations, two different values for C were tested: zero and 10^3, to see how it influenced the times. For **before**(k), we use $k = 10$ and $k = 500$. Finally, **with** was used with $k = 5$, except for the *lazy* version, that was also run with $k = 100$, since it was likely to be affected by k. All other k and $[s]$ parameters do not affect performance.

We performed about 1300 tests in total, each one run on five different examples.

A test for lazy vs merge: Its aim is to determine the average expansion of the trees made by the *lazy* algorithms, and to compare times against full *merge*. The time comparison was also done in the first test, but in this case we selected 10 queries of different complexity, on C code (observe that the queries of the first test are all of one operator only). The idea was to know what is the percentage of expansion in a real query (i.e. with more than one operation involved), where lazy evaluation is likely to behave better, and what performance can we expect in queries involving many operations.

A test for binary vs interpolation search: Its aim is to determine how close is the real distribution to a uniform one, to decide whether interpolation search is a good idea. We tested it for sizes between 10 and 10^4, in each case with ten pairs of flat operands selected from the documents. These case were run as a "−" operation, since being different the operands, the process would involve just search time.

The detailed results of the tests are shown in Appendix C. Figures 6.3 and 6.4 show some typical results.

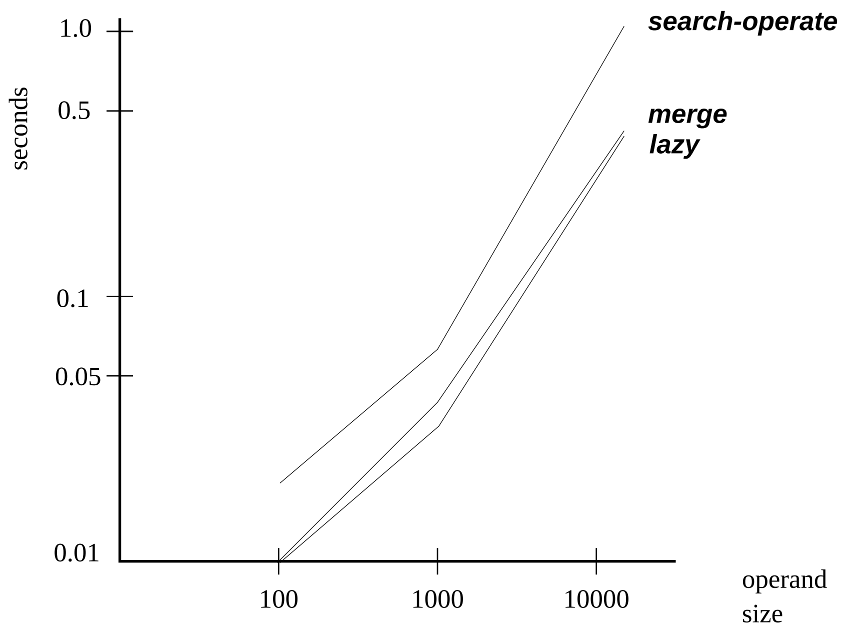


Figure 6.3: Typical times for equal-sized flat operands. Observe that we use a logarithmic scale.

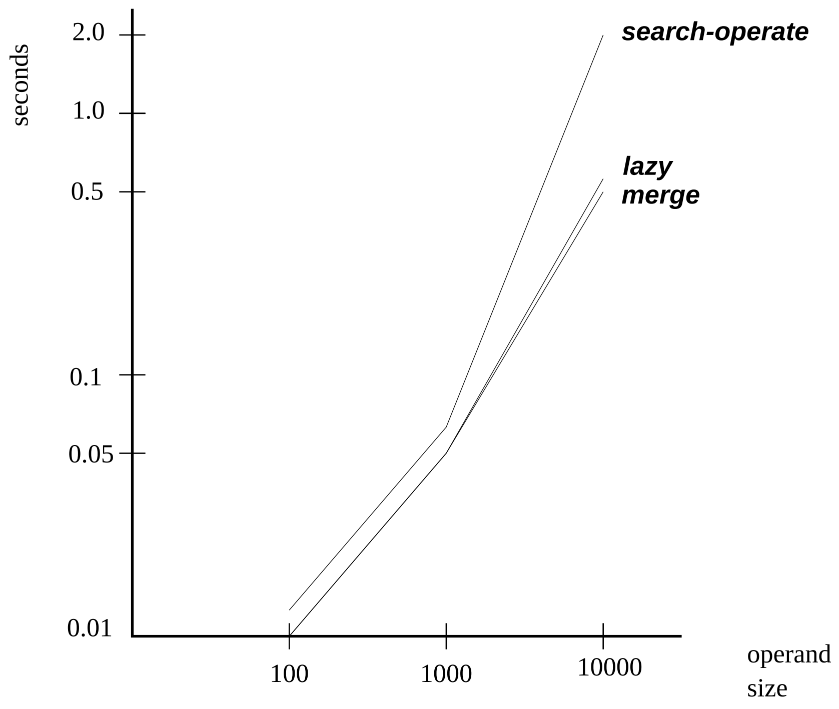


Figure 6.4: Typical times for equal-sized deep operands. Observe that we use a logarithmic scale.

6.2.3 Results

From the results of the tests, we can strongly conclude a number of important facts:

- The *merge* algorithms are by far better than their *search-operate* counterparts. Although both versions are similar for sizes of 10^3 or less, *merge* algorithms are much better for larger sets, for example by a factor between 2 and 3 for operands of size 10^4. This difference cannot be attributed to memory management, since it remains even in tests where no modification of the operands occur.
- The *merge* algorithms are strongly linear in most cases, depending only on the size of the operands. *Search-operate* algorithms, on the other hand, are not linear and get much worse when we jump from 10^3 to 10^4. Moreover, they are extremely unstable, depending a lot on internal details of the structure of operands.
- This unstability is not only bad for interactive activities, but also prevents from drawing reasonable heuristics to apply "12" or "21". The algorithms that allow switching between "12" and "21" in subproblems are even more unstable, and can be even definitely worse than using "12" or "21" alone. This is because they can take the wrong decision not only the first time, but also in subproblems.
- However, this issue is not that important, since the "12" and "21" versions are not too different, even when run on sets of very different size. This is the reason because we define our heuristics by complexity and some intuitive considerations in the definition of the algorithms: it is difficult to make it better, and does not make a big difference.
- The running time of *merge* algorithms is normally easy to predict, except in some cases (especially the C of **before** seems to affect retrieval times in unpredictable ways). For most operations, the times indicate that near 50.000 nodes per second are processed in this machine.
- Although in one-operator queries the *lazy* algorithms are roughly equivalent to full *merge* ones, the *lazy* approach is normally better in more complex queries, but its times are quite unstable. As we may

expect, its performance is very dependent upon how much does it need to expand, and this is close related to the internal structure of trees and the kind of operations we are performing.

- In the real queries tested, the *lazy* algorithm behaved in a very variable way. Its percentage of expansion ranged from 40% to almost 100%, depending on the structure and sizes of the operands and intermediate results.
- The speed of *merge* algorithms in the complex queries was proportional to the sum of the sizes of all internal results. The constant for our machine is between $[1.5 \,..\, 2.5] \times 10^{-5}$ seconds per node, or between $[4 \,..\, 7] \times 10^{4}$ nodes per second. The time for lazy evaluation was normally half the time for *merge*, being a common range a fraction of $[0.25 \,..\, 0.90]$.
- It is important to note another difference between full and lazy evaluation: we implemented modifying versions of full algorithms and generating versions of *lazy* ones. Modifying algorithms are better if the results are a big portion of the operand, generating algorithms are better otherwise. This aspect was not foreseen in the theoretical analysis, but emerges as an important practical consideration, since the results indicate that generating algorithms are much better in practice.
- A deeper structure of operands affects negatively the mean and variance of most operators in all versions, although *merge* algorithms are less affected.
- On the tests run for binary vs interpolation search, interpolation performed much worse than binary search.

These facts raise some general conclusions to learn from the tests:

- We should concentrate on *merge* and *lazy* algorithms and discard the *search-operate* version. This involves more complex algorithms, it is more difficult to analyze, is very unstable and nonlinear (also in practice) and has definitely worse performance. *Merge* is strongly linear, predictable, efficient and simpler.
- A rough estimation for the time taken by *merge* algorithms to process a query whose syntax tree has q nodes is $(2q-1) \times (\textit{average operand size})$. The constant depends on the machine, for example in our case it is $\approx 2 \times 10^{-5}$ seconds per node (50.000 nodes per second).
- *Lazy* operations presents a high variance compared to *merge*, expanding between 40% and 100% of the query and taking between 25% and 90% of the time *merge* needs to retrieve the result. It is probably worth to extend the lazy mechanism to expand only some children of a node, since its advantage is related to how much it has to expand.
- The problem of modifying vs generating algorithms should be separated from full vs lazy evaluation, and deeply studied in the context of query optimization and query plan generation, because they strongly influence practical times. Mixed strategies, such as modifying operations plus garbage collection to avoid processing the nodes to be deleted, should be studied.

Chapter 7

Conclusions and Future Work

7.1 Summary

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 very expressive or inefficient.

Then, we have defined a model for structuring documents and a query language that are expressive and efficiently implementable.

To evaluate expressivity we have compared our model with other known ones, showing which aspects of each one can be represented with the other; and we defined a framework in which to evaluate any similar model, situating known approaches.

To evaluate efficiency we have implemented our model, and analyzed the behavior of our algorithms in terms of both space and time. We also implemented a prototype to draw average measures and to test different alternative solutions to some problems.

7.2 Conclusions

The results of this work are a new model for structuring and querying textual databases which has an efficient implementation and good expressivity, a framework to compare expressivity and an analysis of the expressivity of the new model, an implementation of the operations, and a theoretical and practical study of their behavior in terms of both time and space.

The main idea of this work is that a set-oriented query language based on operations on nearby nodes is very expressive and efficiently implementable. The strong relation imposed between nodes and segments allows us to index the nodes in a tree, based on their left-to-right and including relations. This way, at the moment of operating two sets, we can easily apply divide-and conquer techniques, thus avoiding costly all-against-all matching algorithms such as those studied in [KM92, KM95]. On the other hand, most interesting operations can be carried out by operating nearby nodes.

We developed two main strategies to solve queries. A full-evaluation approach computes the whole set of answers, while a lazy-evaluation one computes only what is needed to obtain the final result. Both strategies are competitive for set-oriented languages. Additionally, lazy evaluation can be adapted to interactive systems that combine querying and navigation, by expanding only the top-level and thereafter expanding the nodes the user wants to see more closely.

The structuring model and operations defined for the language are shown to be competitive in expressivity, getting closer in expressivity to models that have not an efficient implementation. On the other hand, the *merge* and *lazy* algorithms developed show good and predictable performance, both in their analysis and in the tests, what situates this model closer in efficiency to those which have much less expressivity.

While a number of operations can be added to the language keeping the high efficiency, the work shows that some other features cannot be added without significantly complicating the implementation and degrading the performance. For example, more expressivity can be obtained by dissociating nodes and segments (as in [GT87]), or by including non-proximal operations (like semijoin), but at the cost of losing efficiency.

Finally, recall that the proposed language is an operational algebra, not necessarily intended to be accessed by the final user, as the relational algebra is not used by the users of the database. It serves as an intermediate representation of the operations.

See Figure 7.1 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. Note that, as any quantization of concepts this comparison is busjective. Nevertheless, it does give an idea of where our model is.

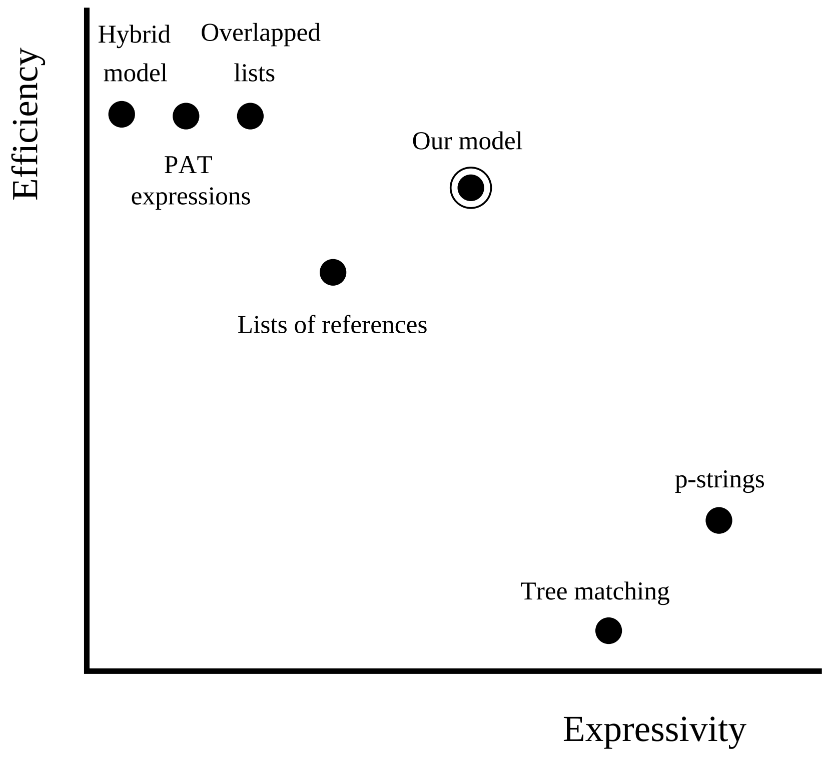


Figure 7.1: A comparison between similar models, regarding both efficiency and expressivity.

This work makes a step in the direction of obtaining a unifying perspective on how a query language for textual databases should be, how much expressive power should it have and how well can it be implemented. All this is necessary to put the emerging area of textual databases in the place it deserves in Computer Science.

7.3 Future Work

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

- To explore the possibilities offered by the data structures in order to find more interesting operators which are efficiently implementable. The capabilities of the proposed data structure are by no means fully exploited. On the other hand, we should find a set of operators which is simple, elegant and expressive, as for example the relational algebra. In this work we focused only in showing which things could be expressed with our approach, but not in a single and simple language.

- To improve the lazy mechanism, by allowing to expand only parts of levels, since as it is, the mechanism expands complete levels and that is not good for shallow trees (and most of them are).
- To implement the proposed techniques for maintaining the index and operands on disk, to study the behavior of different approaches and policies. We have disregarded this problem here, implementing the index and operands in memory.
- To design swapping policies for when the operands do not fit in memory. We have disregarded this problem here, having all operands in memory all the time.
- To design a query parser that generates smart query plans, implementing the proposed techniques (trees, DAGs, and deallocation policies), to experimentally study their behavior. We have not implemented the query parser, but generate manually the query plans.
- Related to the previous is the need to find a good way to avoid processing nodes that must be deleted, and to draw a general heuristic in the use of modifying or generating algorithms.
- To implement parsers and filters to index new types of structuring mechanisms.
- To define a query language suitable for end users, possibly visual, to map onto our operational algebra.
- To enhance the prototype, to convert it into a real application, with management of indices on disk, swapping, query plan generation, etc. An interesting possibility is to use the lazy mechanism as the engine of a system combining set-oriented queries with navigational tools.
- To prove formally the correctness of our algorithms, which we have done only informally.
- To study an extension to our model that allows manipulating the results of queries as new views, as in relational databases.

7.4 Open Problems

Finally, we point out a number of problems raised by this work, which deserve a separate and deep study and attention:

- The design of query optimization techniques for our language. Once the query plan generator is written, there is a large number of possible optimizations, by using algebraic identities that allow transformation of an expression to a cheaper one. Another direction is to make some compound operations atomic, which may be cheaper to implement that way (a typical example is $P - (P\ such\ that...)$). Some ideas and theoretical results about optimization can be found in [CM94].
- An interesting area is the integration between this kind of model and others, such as the relational or the traditional IR ones. This issue has not been considered here, since we focused on the structure problem. See [SDAMZ94] for some ideas on this area. A promising direction to manipulate tuples in our language is to convert all operators of the kind "select nodes of P which are related in such way with a node of Q" to "select pairs (p, q) from $P \times Q$ related in a such way".
- As the research in this area matures, it is possible that a reasonable probabilistic model for the structure of documents emerges, at least for restricted areas. In that case, an average-case study of the behavior of our algorithms in both space and time would be of most interest.
- The generalization of the problem to manage non-hierarchical structures, such as a hypertext network. The design of a good and efficient query language for this case has not been studied, to our knowledge.

- A formal framework in which to compare expressivity is needed. We have done one step forward in that direction in this work, but the long-term goal should be a formal and sound hierarchy like what can be found in the area of formal languages (see [CM95, GPG89] for some examples). Related to this is the problem of finding lower bounds for the complexity of the operations related to structure, in order to know when we have found the best algorithms.
- Finally, we need not only a formal but also a practical standard of expressivity. Currently, a wide diversity of expressivity models exist, and no consensus about what one would like to express. Finding a unified view on this subject is a matter of maturing.

Bibliography

[Ars92] Ars Innovandi, Santiago, Chile. *Search City 1.1. Text Retrieval for Windows Power Users*, 1992.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[BBT92] G. Blake, T. Bray, and F. Tompa. Shortening the OED: Experience with a grammar-defined database. *ACM Transactions on Information Systems*, 10(3):213–232, July 1992.

[BGT91] D. Berg, G. Gonnet, and F. Tompa. The new Oxford English Dictionary project at the University of Waterloo. In A. Zampolli, editor, *Computational Issues in Lexicology and Linguistics, Special Issue in Honour of Bernard Quemada*, Pisa, 1991. Giardiana Editori.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

[Bur92a] F. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348, 1992.

[Bur92b] F. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In N. Belkin, P. Ingwersen, and A. Pejtersen, editors, *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 112–125. ACM Press, June 1992.

[BY94] R. Baeza-Yates. An hybrid query model for full text retrieval systems. Technical Report DCC-1994-2, Department of Computer Science, University of Chile, 1994.

[BYJQ92] R. Baeza-Yates, L. Jara, and G. Quezada. VCC: Automatic animation of C programs. *COMPUGRAPHICS'92, Lisboa, Portugal*, 1992.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM-SIGMOD 1994 Annual Conference on Management of Data*, pages 313–324, 1994.

[CCB95a] 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. An early version can be found as University of Waterloo Computer Science Department Technical Report CS-94-30.

[CCB95b] C. Clarke, G. Cormack, and F. Burkowski. Schema-independent retrieval from heterogeneous structured text. In *Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, Nevada, April 1995.

[CGG$^+$88] B. Char, K. Geddes, G. Gonnet, M. Monagan, and S. Watt. *Maple Reference Manual, 5th Edition*. Waterloo, 1988.

[CM93] M. Consens and A. Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516, 1993. Video presentation summary.

[CM94] M. Consens and T. Milo. Optimizing queries on files. In *Proceedings of the ACM-SIGMOD 1994 Annual Conference on Management of Data*, pages 301–312, 1994.

[CM95] M. Consens and T. Milo. Algebras for querying text regions. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1995. California.

[Coh81] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Con87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.

[Dat95] C. Date. *An Introduction to Database Systems.* The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 6th edition, 1995.

[Dav92] A. Davie. *An introduction to functional programming systems using Haskell*, chapter 7 and 8. Cambridge University Press, 1992. Cambridge Computer Science Texts No. 27.

[DGS86] B. Desai, P. Goyal, and S. Sadri. A data model for use with formatted and textual data. *Journal of the American Society for Information Science*, 37(3):158–165, 1986.

[Dig91] Digital Equipment Corporation, Maynard, Massachusetts. *CDA - DDIF Technical Specification*, June 1991. Order Number: AA-PHK3A-TE.

[DWL92] S. Deerwester, K. Waclena, and M. LaMar. A textual object management system. In N. Belkin, P. Ingwersen, and A. Pejtersen, editors, *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 126–139. ACM Press, June 1992.

[Faw89] H. Fawcett. *PAT 3.3 User's Guide.* UW Centre for the New OED and Text Research, University of Waterloo, 1989.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1992.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability.* W. Freeman and Company, San Francisco, 1979.

[Gol90] C. Goldfarb. *The SGML Handbook.* Oxford, 1990.

[Gon87] G. Gonnet. Examples of PAT applied to the Oxford English Dictionary. Technical Report OED-87-02, UW Centre for the New OED and Text Research, University of Waterloo, 1987.

[GPG89] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. In *Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data*, pages 263–272, 1989.

[GT87] G. Gonnet and F. Tompa. Mind Your Grammar: a new approach to modelling text. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 339–346, 1987.

[HK87] R. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[ISO86] International Standards Organization. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*, 1986. ISO 8879-1986.

[Kil92] P. Kilpeläinen. Tree matching problems with applications to structured text databases. Technical Report A-1992-6, Department of Computer Science, University of Helsinki, Helsinki, Finland, November 1992.

[KL89] W. Kim and F. Lochovski, editors. *Object-Oriented Concepts, Databases and Applications.* Addison-Wesley, Reading, Massachusetts, 1989.

[KM92] P. Kilpeläinen and H. Mannila. Grammatical tree matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 162–174. Springer-Verlag, 1992.

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In R. Korfhage, E. Rasmussen, and P. Willett, editors, *SIGIR '93 — Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, New York, 1993. ACM Press.

[KM95] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, April 1995.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs NJ, 1978.

[Lam86] L. Lamport. LaTeX*: A Document Preparation System.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Loe94] A. Loeffen. Text databases: A survey of text models and systems. *ACM SIGMOD'94 Record*, 23(1):97–106, March 1994.

[Mac90] I. MacLeod. Storage and retrieval of structured documents. *Information Processing & Management*, 26(2):197–208, 1990.

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

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, CA, January 1990.

[RTW93] D. Raymond, F. Tompa, and D. Wood. Markup reconsidered. Technical Report OED-93-01, UW Centre for the New Oxford English Dictionary, April 1993.

[Sal86] G. Salton. Another look at automatic text retrieval systems. *Communications of the ACM*, 29:648–656, 1986.

[SDAMZ94] R. Sacks-Davis, T. Arnold-Moore, and J. Zobel. Database systems for structured documents. In *Proceedings of the International Symposium on Advanced Database Technologies and Their Implementation, ADTI'94*, pages 272–283, Nara, Japan, 1994.

[SM83] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[SM89] G. Salton and M. McGill. *Automatic text processing.* Addison-Wesley, Reading, Massachusetts, 1989.

[SSL$^+$83] M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. Document processing in a relational database system. *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.

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

[SW87] J. Smith and S. Weiss. Formatting texts accessed randomly. *Software Practice and Experience*, 17(1):5–16, January 1987.

[TSM91] J. Tague, A. Salminen, and C. McClellan. Complete formal model for information retrieval systems. In *Proceedings of the 14th Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 14–20, 1991.

Appendix A

Formal Syntax and Semantics

We first define the syntax of our expressions ($Expr$) by an annotated abstract syntax. In this definition, we use N as the set of natural numbers, Z as the integers, M as the set of pattern-matching expressions, S 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 views. We could address this problem formally, but it would be cumbersome, so we prefer to indicate at the right side of each alternative, the conditions on the views of the operands and the view of the result, between brackets. The view of the result is expressed as a function $\tau : Expr \to \mathcal{V}$.

$$
\begin{array}{rcl}
Expr & \longrightarrow & \mathbf{View}(V)\ [V \in \mathcal{V} - \{V_t\}, \tau = V] \\
& | & \mathbf{Constr}(c)\ [c \in \mathcal{C} - \{C_t\}, \tau = V / c \in C_V] \\
& | & \mathbf{Match}(m)\ [m \in M, \tau = V_t] \\
& | & (E_1\ \mathbf{collapse/subtract/intersect}\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) = V_t] \\
& | & \mathbf{join/complement}(E)\ [\tau = \tau(E) = V_t] \\
& | & \mathbf{move}(i,j)\ E_1\ (E_2))\ [i,j \in Z, \tau = \tau(E_1) = V_t] \\
& | & (E_1\ \mathbf{to}\ E_2\ (E_3))\ [\tau = \tau(E_1) = \tau(E_2) = V_t] \\
& | & \mathbf{toplevel}(E)\ [\tau = V_t] \\
& | & (E_1\ +\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t] \\
& | & (E_1\ -\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t] \\
& | & (E_1\ \mathbf{is}\ E_2)\ [\tau = \tau(E_1) = \tau(E_2) \neq V_t] \\
& | & (E_1\ \mathbf{same}\ E_2)\ [\tau = \tau(E_1)] \\
& | & (E_1\ \mathbf{with}(k)\ E_2)\ [k \in N, \tau = \tau(E_1)] \\
& | & (E_1\ \mathbf{withbegin/withend}(k)\ E_2)\ [k \in N, \tau = \tau(E_1) \neq \tau(E_2)] \\
& | & (E_1\ \mathbf{in}\ E_2)\ [\tau = \tau(E_1)] \\
& | & (E_1\ \mathbf{beginin/endin}\ E_2)\ [\tau = \tau(E_1) \neq \tau(E_2)] \\
& | & ([s]\ E_1\ \mathbf{in}\ E_2)\ [s \in S, \tau = \tau(E_1)] \\
& | & ([s]\ E_1\ \mathbf{beginin/endin}\ E_2)\ [s \in S, \tau = \tau(E_1) \neq \tau(E_2)] \\
& | & (E_1\ \mathbf{parent}(k)\ E_2)\ [k \in N, \tau = \tau(E_1) = \tau(E_2) \neq V_t] \\
& | & ([s]\ E_1\ \mathbf{child}\ E_2)\ [s \in S, \tau = \tau(E_1) = \tau(E_2) \neq V_t] \\
& | & (E_1\ \mathbf{after/before}\ E_2\ (E_3))\ [\tau = \tau(E_1)] \\
& | & (E_1\ \mathbf{after/before}(k)\ E_2\ (E_3))\ [k \in N, \tau = \tau(E_1)]
\end{array}
$$

Now, the function $\mathcal{I}$ is defined inductively as:

- $\mathcal{I}(\mathbf{View}(V)) = N_V$.
- $\mathcal{I}(\mathbf{Constr}(c)) = \{x \in \mathcal{N} / Constr(x) = c\}$.
- Suppose m is a pattern-matching expression, whose result is a set of segments $(a_1, b_1)..(a_k, b_k)$. Then, $\mathcal{I}(m) = \{t_{a_i,b_i} / i \in [1..k]\}$.

- $\mathcal{I}(P$ **collapse** $Q) = \{t_{a_1,b_n} / \exists t_{a_1,b_1} ... t_{a_n,b_n} \in \mathcal{I}(P) \cup \mathcal{I}(Q) / (\forall i, b_i \leq a_{i+1}) \wedge \nexists t_{x,y} \in \mathcal{I}(P) \cup \mathcal{I}(Q) - \{t_{a_1,b_1} ... t_{a_n,b_n}\} / (x,y) \cap (a_1, b_n) \neq \emptyset\}$.
- $\mathcal{I}($**join** $P) = \{t_{a_1,b_n} / \exists t_{a_1,b_1} ... t_{a_n,b_n} \in \mathcal{I}(P) / (\forall i, a_{i+1} = b_i + 1) \wedge \nexists t_{x,y} \in \mathcal{I}(P) / (x = b_n + 1 \vee y = a_1 - 1)\}$.
- $\mathcal{I}(P$ **intersect** $Q) = \{t_{a,b} / \exists t_{x,y} \in \mathcal{I}(P), t_{z,w} \in \mathcal{I}(Q) / (x,y) \cap (z,w) = t_{a,b} \neq \emptyset\}$.
- $\mathcal{I}($**complement** $P) = \{t_{a,b} / a \leq b \ \wedge \ \exists t_{x,a-1}, t_{b+1,y} \in \mathcal{I}(P) \cup \{t_{0,0}, t_{T+1,T+1}\}\}$.
- $\mathcal{I}(P$ **minus** $Q) = \mathcal{I}(P$ **intersect** $($**complement** $Q))$.
- $\mathcal{I}($**toplevel** $P) = \{t_{a,b} / \exists x \in maxim(\mathcal{I}(P)) / Segm(x) = (a,b)\}$. Here we use $maxim : \wp(\mathcal{N}) \rightarrow \wp(\mathcal{N})$, defined as $maxim(X) = \{x \in X / \ \nexists x' \in X / x' \supset x\}$ (on chains, it returns singletons).
- $\mathcal{I}($**move**$(n,m) \ P \ (C)) = \{t_{a',b'} / a' \leq b' \ \wedge \ \exists p \in \mathcal{I}(P) / Segm(p) = (a,b) \ \wedge \ (a+n, b+m) \cap Segm(minim(z \in \mathcal{I}(C) / p \subset z)) = (a', b')\}$. Here we use $minim : \wp(\mathcal{N}) \rightarrow \wp(\mathcal{N})$, defined as $minim(X) = \{x \in X / \ \nexists x' \in X / x \supset x'\}$ (on chains, it returns singletons, that we interpret here as their only element).
- $\mathcal{I}(P$ **to** $Q \ (C)) = minim\{t_{a,d} / a \leq d \ \wedge \ \exists x \in \mathcal{I}(P) / Segm(x) = (a,b) \ \wedge \ \exists y \in \mathcal{I}(Q) / Segm(y) = (c,d) \ \wedge \ minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\}$.
- $\mathcal{I}(P \ + \ Q) = \mathcal{I}(P) \cup \mathcal{I}(Q)$.
- $\mathcal{I}(P \ - \ Q) = \mathcal{I}(P) - \mathcal{I}(Q)$.
- $\mathcal{I}(P$ **is** $Q) = \mathcal{I}(P) \cap \mathcal{I}(Q)$.
- $\mathcal{I}(P$ **same** $Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / Segm(x) = Segm(y)\}$.
- $\mathcal{I}(P$ **with**$(k) \ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / y \subset x\}| \geq k\}$.
- $\mathcal{I}(P$ **withbegin**$(k) \ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / From(y) \in Segm(x)\}| \geq k\}$.
- $\mathcal{I}(P$ **withend**$(k) \ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / To(y) \in Segm(x)\}| \geq k\}$.
- $\mathcal{I}(P$ **in** $Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / x \subset y\}$.
- $\mathcal{I}(P$ **beginin** $Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / From(x) \in Segm(y)\}$.
- $\mathcal{I}(P$ **endin** $Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / To(x) \in Segm(y)\}$.
- $\mathcal{I}([s] \ P$ **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 the segment of a node in the segments of 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 $\mathcal{Z}_y$. Finally, $\mathcal{Z}_y = \{x \in P / x \subset y \ \wedge \ x \in maxim(z \in P / z \subset y \ \vee \ y \not\subset z)\}$.
- $\mathcal{I}([s] \ P$ **beginin** $Q) = \bigcup_{y \in \mathcal{I}(Q)} \{x \in \mathcal{Z}_y / \mathcal{S}(s, x, \mathcal{Z}_y)\}$. Here, $\mathcal{Z}_y = \{x \in P / From(x) \in Segm(y) \wedge \ x \in maxim(z \in P / Segm(z) \not\supset Segm(y))\}$.
- $\mathcal{I}([s] \ P$ **endin** $Q) = \bigcup_{y \in \mathcal{I}(Q)} \{x \in \mathcal{Z}_y / \mathcal{S}(s, x, \mathcal{Z}_y)\}$. Here, $\mathcal{Z}_y = \{x \in P / To(x) \in Segm(y) \wedge \ x \in maxim(z \in P / Segm(z) \not\supset Segm(y))\}$.
- $\mathcal{I}(P$ **parent**$(k) \ Q) = \{x \in \mathcal{I}(P) / |\{y \in \mathcal{I}(Q) / x \longrightarrow y\}| \geq k\}$.
- $\mathcal{I}([s] \ P$ **child** $Q) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / y \longrightarrow x \wedge \mathcal{S}(s, x, \{z \in \mathcal{N} / y \longrightarrow z\})\}$.
- $\mathcal{I}(P$ **after**$(k) \ Q \ (C)) = \{x \in \mathcal{I}(P) / \exists y \in \mathcal{I}(Q) / 0 < From(x) - To(y) \leq k \ \wedge \ minim(\{z \in \mathcal{I}(C) / x \subset z\}) = minim(\{z \in \mathcal{I}(C) / y \subset z\})\}$.

- $\mathcal{I}(P$ **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 view, this criterion gives exactly one node.

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

- $\mathcal{I}(P$ **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.

Appendix B

Translation Formulas

B.1 Hybrid Model

We depict here functions $\mathcal{D}$ and $\mathcal{M}$, to translate from the hybrid model to ours, and the function $\mathcal{R}$, to do the converse. We begin with the parallel definition of $\mathcal{D}$ and $\mathcal{M}$.

- Matching expressions m: we do not enter into details here. They return a set of size-1 text segments. Although we do not define $\mathcal{M}(m)$, it holds $\mathcal{D}(m) = DV$ **with** $\mathcal{M}(m)$.
- $P\ +/\ or\ Q$ makes the union of match points and documents of P and Q:
  - $\mathcal{M}(P\ +/\ or\ Q) = \mathcal{M}(P)$ **collapse** $\mathcal{M}(Q)$.
  - $\mathcal{D}(P\ +/\ or\ Q) = \mathcal{D}(P) + \mathcal{D}(Q)$.
- $P - Q$ subtracts the matches of Q from the matches of P. The resulting D component are documents which contain some resulting match.
  - $\mathcal{M}(P - Q) = \mathcal{M}(P)$ **subtract** $\mathcal{M}(Q)$.
  - $\mathcal{D}(P - Q) = DV$ **with** $\mathcal{M}(P - Q)$.
- $P \& Q$ intersects the matches of Q with the matches of P. The resulting D component are documents which contain some resulting match.
  - $\mathcal{M}(P \& Q) = \mathcal{M}(P)$ **same** $\mathcal{M}(Q)$.
  - $\mathcal{D}(P \& Q) = DV$ **with** $\mathcal{M}(P \& Q)$.
- $\sim P$ takes all match points except those present in P. The resulting D component are documents which contain some resulting match.
  - $\mathcal{M}(\sim P) =$ "*" **subtract** $\mathcal{M}(P)$. We assume here that we have in the matching sublanguage a query "*" to retrieve all points of the text (which is quite problable).
  - $\mathcal{D}(\sim P) = DV$ **with** $\mathcal{M}(\sim P)$.
- $P\ and\ Q$ intersects the documents of P and Q. The resuling match points are those which appear in P or in Q (inside the resulting documents).
  - $\mathcal{M}(P\ and\ Q) = (\mathcal{M}(P)$ **collapse** $\mathcal{M}(Q))$ **in** $\mathcal{D}(P\ and\ Q)$.
  - $\mathcal{D}(P\ and\ Q) = \mathcal{D}(P)$ **is** $\mathcal{D}(Q)$.
- $P\ butnot\ Q$ subtracts the document of Q from the documents of P. The resulting matches are those of the remaining P documents.

  - $\mathcal{M}(P\ butnot\ Q) = \mathcal{M}(P)$ **subtract** $(\mathcal{M}(P)$ **in** $\mathcal{D}(Q))$.
  - $\mathcal{D}(P\ butnot\ Q) = \mathcal{D}(P) - \mathcal{D}(Q)$.
- *not* P returns all documents except those included in P, and an empty set of matches.
  - $\mathcal{M}(not\ P) =$ "*" **subtract** "*", or any mechanism to obtain an empty text query.
  - $\mathcal{D}(not\ P) = DV - \mathcal{D}(P)$.
- $P\ in\ f$ restricts the match points of P to those appearing inside a field f. The set of documents is restricted to those containing some resulting match point.
  - $\mathcal{M}(P\ in\ f) = (\mathcal{M}(P)$ **in** $f_1)$ **collapse** ... **collapse** $(\mathcal{M}(P)$ **in** $f_r)$.
  - $\mathcal{D}(P\ in\ f) = \mathcal{D}(P)$ **with** $\mathcal{M}(P\ in\ f)$.
- $f\ with\ P$ returns matches corresponding to the beginnings of fields f containing some match of P, and the documents containing those beginnings.
  - $\mathcal{M}(f\ with\ P) =$ (**beginnings** $(f_1$ **with** $\mathcal{M}(P)))$ **collapse** ... **collapse** (**beginnings** $(f_r$ **with** $\mathcal{M}(P)))$.
  - $\mathcal{D}(f\ with\ P) = DV$ **with** $\mathcal{M}(f\ with\ P)$.

This completes the representation, except for *some*, whose syntax is $some_k(q_1, ..., q_m)$, with $1 \leq k \leq m$ and q_i queries. It means that a document is retrieved only if at least k of the m queries retrieve it. The only simulation we can offer for *some* is of length $k\binom{m}{k} = O(m2^m)$, by rewriting

$$some_k(q_1, ..., q_m) = or_{q_{i_1} \neq ... \neq q_{i_k} \in \{q_1, ..., q_m\}}(q_{i_1}\ and\ ...\ and\ q_{i_k})$$

The definition of the converse function, $\mathcal{R}$, follows.

- $\mathcal{R}(\mathbf{Constr}(c)) = c\ with$ "*".
- $\mathcal{R}(\mathbf{View}(V)) = (c_1\ with$ "*"$) + ... + (c_r\ with$ "*"$)$, where $C_V = \{c_1, ..., c_r\}$.
- $\mathcal{R}(\mathbf{Match}(m))$. We left this unspecified.
- $\mathcal{R}(P + Q) = \mathcal{R}(P) + \mathcal{R}(Q)$.
- $\mathcal{R}(P - Q) = \mathcal{R}(P) - \mathcal{R}(Q)$.
- $\mathcal{R}(P$ **is/same** $Q) = \mathcal{R}(P)\ \&\mathcal{R}(Q)$. Recall that since no two segments begin at the same position, **is** = **same**.
- $\mathcal{R}(P$ **beginin** $Q) = \mathcal{R}(P)\ in\ c$. This works only if $Q = \mathbf{Constr}(c)$ (it is not compositional). If $Q = \mathbf{Constr}(c_1) + ... + \mathbf{Constr}(c_r)$, it can be translated into $(\mathcal{R}(P)\ in\ c_1) + ... + (\mathcal{R}(P)\ in\ c_r)$.
- $\mathcal{R}(P$ **in/endin** $Q)$ can be translated as before, with the additional constraint that the hierarchy must be strict.
- $\mathcal{R}([s]\ P$ **in/beginin/endin** $Q)$ cannot be represented.
- $\mathcal{R}(P$ **parent**/$[s]$ **child** $Q)$: the concept does not exist.
- $\mathcal{R}(P$ **withbegin** $Q) = c\ with\ \mathcal{R}(Q)$. This works only if $P = \mathbf{Constr}(c)$ and $k = 1$. Again, if $Q = \mathbf{Constr}(c_1) + ... + \mathbf{Constr}(c_r)$, it can be translated into $(c_1\ with\ \mathcal{R}(Q)) + ... + (c_r\ with\ \mathcal{R}(Q))$.
- $\mathcal{R}(P$ **with/withend** $Q)$ can be translated as before, with the additional constraint that the hierarchy must be strict.
- $\mathcal{R}(P$ **after/before**$(k)\ Q\ (C)) = \mathcal{R}(P)\ after/before.k\ \mathcal{R}(Q)$. This only works if $C = none$. Notice also that *after/before* measure from segment start to segment start, so the semantics is in fact different.
- $\mathcal{R}(P$ **after/before** $Q\ (C))$ cannot be represented.

B.2 PAT Expressions

We depict here the function $\mathcal{I}$, to translate from PAT expressions to our model, and the function $\mathcal{P}$, to do the converse. We begin with the definition of $\mathcal{I}$.

- Matching expressions m: we disregard this issue here, assuming we get a set of match points (size-1 text segments).
- reg is the set of all areas corresponding to reg, thus $\mathcal{I}(reg) =$ **toplevel** (V_{reg}).
- P *within* Q is the set of areas of P whose start point is in some area of Q, thus $\mathcal{I}(P \textit{ within } Q) = \mathcal{I}(P)$ **beginin** $\mathcal{I}(Q)$.
- P *including.n* Q is the set of areas of P including at least n start points from areas of Q, thus $\mathcal{I}(P \textit{ including.n } Q) = \mathcal{I}(P)$ **withbegin**(n) $\mathcal{I}(Q)$.
- $P + Q$ makes the union of areas, but if they overlap, all areas are converted to their start points, thus $\mathcal{I}(P + Q) = \mathcal{I}(P)$ **collapse** $\mathcal{I}(Q)$ if it does not cause overlaps, else (**beginnings** $\mathcal{I}(P)$) **collapse** (**beginnings** $\mathcal{I}(Q)$).
- $P - Q$ makes set difference, thus $\mathcal{I}(P - Q) = \mathcal{I}(P)$ **subtract** $\mathcal{I}(Q)$.
- $P \wedge Q$ makes set intersection, thus $\mathcal{I}(P \wedge Q) = \mathcal{I}(P)$ **same** $\mathcal{I}(Q)$.
- P *fby.n* Q is the set of areas of P *starting* at most n points before the start of an area of Q, thus $\mathcal{I}(P \textit{ fby.n } Q) = \mathcal{I}(P)$ **before**(n) $\mathcal{I}(Q)$ $(none)$, except because we count from the end points of P, not from starts. In fact, the same [ST92] recognizes that this is not so good. If $.n$ is not present, we use **before** (∞).
- P *near.n* Q is as *fby*, but the Q area can be before or after P, thus $\mathcal{I}(P \textit{ near.n } Q) = (\mathcal{I}(P)$ **before**(n) $\mathcal{I}(Q)$ $(none))$ **collapse** $(\mathcal{I}(P)$ **after**(n) $\mathcal{I}(Q)$ $(none))$, with the same special cases as before.
- P *not op* Q (a metarrule for all operators *op* allowing a *not* clause) retrieves the elements of P not retrieved by P *op* Q, thus $\mathcal{I}(P \textit{ not op } Q) = \mathcal{I}(P)$ **subtract** $\mathcal{I}(P \textit{ op } Q)$.

The definition of the converse function, $\mathcal{P}$, follows.

- $\mathcal{P}(\textbf{Constr}(c)) = reg_c$.
- $\mathcal{P}(\textbf{View}(V)) = reg_{c_1} + ... + reg_{c_r}$, where $C_V = \{c_1, ..., c_r\}$.
- $\mathcal{P}(\textbf{Match}(m))$, we disregard this point here.
- $\mathcal{P}(P + Q) = \mathcal{P}(P) + \mathcal{P}(Q)$.
- $\mathcal{P}(P - Q) = \mathcal{P}(P) - \mathcal{P}(Q)$.
- $\mathcal{P}(P \textbf{ is/same } Q) = \mathcal{P}(P) \wedge \mathcal{P}(Q)$. Since regions and their segments are exactly the same, **is** = **same**.
- $\mathcal{P}(P \textbf{ beginin } Q) = \mathcal{P}(P) \textit{ within } \mathcal{P}(Q)$.
- $\mathcal{P}(P \textbf{ in/endin } Q)$ is the same as before, but only works if the views do not overlap.
- $\mathcal{P}([s]\ P \textbf{ in/beginin/endin } Q)$ cannot be represented.
- $\mathcal{P}(P \textbf{ withbegin}(k)\ Q) = \mathcal{P}(P) \textit{ including.k } \mathcal{P}(Q)$.
- $\mathcal{P}(P \textbf{ with/withend } Q)$ is the same as before, but only works if the views do not overlap.
- $\mathcal{P}(P \textbf{ parent/}[s] \textbf{ child } Q)$: the concept does not exist.

- $\mathcal{P}(P$ **before**(k) Q $(C)) = \mathcal{P}(P)$ $fby.k$ $\mathcal{P}(Q)$, only if $C = none$. The same comment about measuring from start to start instead of from end to start.
- $\mathcal{P}(P$ **after**(k) Q $(C))$ cannot be translated (observe that $near - fby$ does not work, since a P node that classifies because it has a Q node before it and another after it should be included, but it is eliminated by $near - fby$).
- $\mathcal{P}(P$ **after/before** Q $(C))$ cannot be translated.

B.3 Overlapped Lists

We depict here the function $\mathcal{H}$, to translate from overlapped lists to our model, and the function $\mathcal{W}$, to do the converse. We begin with the definition of $\mathcal{H}$.

- Matching expressions m: we disregard this issue here, assuming we get a set of disjoint segments.
- reg is the set of all areas corresponding to reg, thus $\mathcal{H}(reg) =$ **toplevel** (V_{reg}).
- $P \lhd Q$ is the set of areas of P included in some area of Q, thus $\mathcal{H}(P \lhd Q) = \mathcal{H}(P)$ **in** $\mathcal{H}(Q)$.
- $P \ntriangleleft Q$ is the opposite, thus $\mathcal{H}(P \ntriangleleft Q) = \mathcal{H}(P)$ **subtract** $(\mathcal{H}(P)$ **in** $\mathcal{H}(Q))$.
- $P \rhd Q$ is the set of areas of P including in some area of Q, thus $\mathcal{H}(P \rhd Q) = \mathcal{H}(P)$ **with** $\mathcal{H}(Q)$.
- $P \ntriangleright Q$ is the opposite, thus $\mathcal{H}(P \ntriangleright Q) = \mathcal{H}(P)$ **subtract** $(\mathcal{H}(P)$ **with** $\mathcal{H}(Q))$.
- $P \bigtriangledown Q$ is the union of the sets, eliminating nesting (we also collapse segments when overlaps result), thus $\mathcal{H}(P \bigtriangledown Q) = \mathcal{H}(P)$ **collapse** $\mathcal{H}(Q)$.
- $P \diamond Q$ are segments that extend from P to Q elements such that P is before Q, selecting the minimal ones. Thus, $\mathcal{H}(P \diamond Q) = \mathcal{H}(P)$ **to** $\mathcal{H}(Q)$ $(none)$.
- $P \bigtriangleup Q$ is $(P \diamond Q) + (Q \diamond P)$, but it causes overlaps. So we represent it by collapsing: $\mathcal{H}(P \bigtriangleup Q) = (\mathcal{H}(P)$ **to** $\mathcal{H}(Q)$ $(none))$ **collapse** $(\mathcal{H}(Q)$ **to** $\mathcal{H}(P)$ $(none))$.
- $n \bigtriangleup (A_0, ..., A_{m-1})$ is proposed as a generalization of $\bigtriangleup$ and $\bigtriangledown$. For each set of matches of $A_0 \bigtriangledown A_1 ... \bigtriangledown A_{m-1}$, it selects any possible subset of size m and returns the concatenation of its elements. Although the only representation we could do for this operator is exponential in length, observe that since we cannot represent the overlaps, our answer is the same as for $A_0 \bigtriangledown A_1 ... \bigtriangledown A_{m-1}$.
- n *words* returns all segments formed by n words. This query obviously produces overlapping segments for $n > 1$, and cannot be represented. It makes no sense to represent all the segments by collapsing them. This query is used to restrict the size and distance of other queries.

The definition of the converse function, $\mathcal{W}$, follows.

- $\mathcal{W}($**Constr**$(c)) = reg_c$. It cannot return nested elements, if there are nested elements, the innermost are returned.
- $\mathcal{W}($**View**$(V)) = reg_{c_1} \bigtriangledown ... \bigtriangledown reg_{c_r}$, where $C_V = \{c_1, ..., c_r\}$. The same observation as before.
- $\mathcal{W}($**Match**$(m))$, we disregard this point here.
- $\mathcal{W}(P + Q) = \mathcal{W}(P) \bigtriangledown \mathcal{W}(Q)$. Only minimal segments are returned.
- $\mathcal{W}(P - Q)$ cannot be expressed, except in very special cases (with the help of $\ntriangleleft$ and $\ntriangleright$).

- $\mathcal{W}(P$ **is/same** $Q) = (\mathcal{W}(P) \lhd \mathcal{W}(Q)) \rhd \mathcal{W}(Q)$. This works thanks to not having nested elements in Q. Observe that **is** = **same** in this model, since nodes are just segments. If two nodes could have the same segment, **is** would not be expressible.

- $\mathcal{W}(P$ **in** $Q) = \mathcal{W}(P) \lhd \mathcal{W}(Q)$.

- $\mathcal{W}(P$ **beginin/endin** $Q)$ is the same as before, but only works if the elements do not overlap.

- $\mathcal{W}([s]\ P$ **in/beginin/endin** $Q)$ cannot be represented.

- $\mathcal{W}(P$ **with**$(k)\ Q) = \mathcal{W}(P) \rhd (\mathcal{W}(Q)\diamond...\diamond\mathcal{W}(Q))$ (k times).

- $\mathcal{W}(P$ **withbegin/withend** $Q)$ is the same as before, but only works if the elements do not overlap.

- $\mathcal{W}(P$ **parent**/$[s]$ **child** $Q)$: the concept does not exist.

- $\mathcal{W}(P$ **after/before** $Q\ (C)) = (\mathcal{W}(P) \diamond \mathcal{W}(Q)) \lhd \mathcal{W}(C)$. This works because there is no nesting in C.

- $\mathcal{W}(P$ **after/before**$(k)\ Q\ (C)) = ((\mathcal{W}(P) \diamond \mathcal{W}(Q)) \lhd k\ words) \lhd \mathcal{W}(C)$. *words* are like our symbols, can be words, characters, etc. The same comment as before about nesting in C.

B.4 Lists of References

We depict here the function $\mathcal{X}$, to translate from lists of references to our model, and the function $\mathcal{Y}$, to do the converse. We begin with the definition of $\mathcal{X}$. We use a simplified version of the syntax presented in [Mac91], which is too complex.

- Text matching expressions can be used to restrict nodes to retrieve. We do not address this issue here. They are translated into our text matching expressions in some way.

- *constr* is a constructor name, thus $\mathcal{X}(constr) = [1..last]$ **Constr**$(constr)$ **in** V (recall that V is the name of the only hierarchy we have). The positional inclusion is used to select the top-level components.

- *constrlist.constr* allows to traverse a path in the hierarchy. There are two possibilities:

  - If *constr* can be parent of the last element in *constrlist*, then $\mathcal{X}(constrlist.constr) = [1..last]$ $(V$ **parent** $\mathcal{X}(constrlist))$ **in** V.
  - If *constr* can be a child of the last element in *constrlist*, then $\mathcal{X}(constrlist.constr) =$ **Constr**$(constr)$ **child** $\mathcal{X}(constrlist)$.

  The two situations are not exclussive, but the paper says that things must be arranged in the indexing to avoid ambiguities.

- **any** $n\ P$, when not used after **having**, selects the n first components of P, thus $\mathcal{X}($**any** $n\ P) = [1..n]\ P$ **in Constr**$(archive)$. *archive* is a constructor related to storage organization, and all answers of a query are from the same *archive*. In some situations, the positions must be measured from within another constructor (this can be syntactically determined), chich can also be represented by replacing *archive*.

- P **union** Q makes the union of both sets, if they are from the same constructor, else it is the empty set (this can be syntactically determined). If they are from the same constructor, then $\mathcal{X}(P$ **union** $Q) = \mathcal{X}(P) + \mathcal{X}(Q)$, else any way to obtain an empty set can be used (e.g. $V - V$).

- P **intersection** Q makes the intersection of both sets, so $\mathcal{X}(P$ **intersection** $Q) = \mathcal{X}(P)$ **is** $\mathcal{X}(Q)$.

- P **difference** Q takes the difference, with the same comments about different constructors. $\mathcal{X}(P$ **difference** $Q) = \mathcal{X}(P) - \mathcal{X}(Q)$.

- P **of** Q selects elements of P which descend from a Q element, keeping only the top-level results. Thus, $\mathcal{X}(P$ **of** $Q) = [1..last](\mathcal{X}(P)$ **in** $\mathcal{X}(Q))$ **in** V.
- P **from** Q selects the elements of P that are children of an element of Q. Thus, $\mathcal{X}(P$ **from** $Q) = \mathcal{X}(P)$ **child** $\mathcal{X}(Q)$.
- P **where** *cond* selects elements of P satisfying *cond*. That *cond* can refer to the element itself or to a descendant. Thus,
  - $\mathcal{X}(P$ **where** $(c_1$ **and** $c_2)) = \mathcal{X}((P$ **where** $c_1)$ **where** $c_2)$
  - $\mathcal{X}(P$ **where** $(c_1$ **or** $c_2)) = \mathcal{X}(P$ **where** $c_1) + \mathcal{X}(P$ **where** $c_2)$
  - $\mathcal{X}(P$ **where** $(Q$ **in** $constr)) = \mathcal{X}(P)$ **with** (**Constr**$(constr)$ **with** $\mathcal{X}(Q)$). Here Q can be a text matching expression.
  - $\mathcal{X}(P$ **where** $(Q$ **notin** $constr)) = \mathcal{X}(P) - (\mathcal{X}(P)$ **with** (**Constr**$(constr)$ **with** $\mathcal{X}(Q)$)). Here Q can be a text matching expression.
- P **having any** n Q selects elements of P including at least n elements of Q, thus $\mathcal{X}(P$ **having any** n $Q) = \mathcal{X}(P)$ **with**(k) $\mathcal{X}(Q)$.
- P **having all** Q selects elements of P such that all *constr* included in them are in Q. Here, *constr* is the constructor of the elements in Q (which can be syntactically determined). Thus, $\mathcal{X}(P$ **having all** $Q) = \mathcal{X}(P) - (\mathcal{X}(P)$ **with** (**Constr**$(constr) - \mathcal{X}(Q)$)).
- P **having all** n Q also restricts P elements to include exactly n elements of Q, thus $\mathcal{X}(P$ **having all** n $Q) = \mathcal{X}(P$ **having all** $Q)$ **is** $((\mathcal{X}(P)$ **with**(n) $\mathcal{X}(Q)) - (\mathcal{X}(P)$ **with**$(n+1)$ $\mathcal{X}(Q)))$.
- The same for **not having**, thus $\mathcal{X}(P$ **not having ...** $Q) = \mathcal{X}(P) - \mathcal{X}(P$ **having...** $Q)$.
- P @ (pos) selects *pos*-th elements of P. The language of positions is similar to ours, so $\mathcal{X}(P$ @ $(pos)) = [pos]$ $\mathcal{X}(P)$ **in Constr**$(archive)$. *archive* is as before, and again can be replaced by another constructor.

Now the converse. Some restrictions apply here: we can represent only one hierarchy, we can represent only the top-level of the answers, and we cannot merge elements from different constructors. We define now the $\mathcal{Y}$ function. Observe that **beginin**, **endin**, **withbegin** and **withend** are not applicable, since only one hierarchy exists.

- Text pattern matching expressions are translated in some way that we do not cover here, but in this model text matches can only be used to restrict other answers, not directly returned.
- $\mathcal{Y}(P + Q) = \mathcal{Y}(P)$ **union** $\mathcal{Y}(Q)$, but only works if P and Q are from the same constructor and the result has no nested components.
- $\mathcal{Y}(P - Q) = \mathcal{Y}(P)$ **difference** $\mathcal{Y}(Q)$.
- $\mathcal{Y}(P$ **is** $Q) = \mathcal{Y}(P)$ **intersection** $\mathcal{Y}(Q)$.
- $\mathcal{Y}(P$ **same** $Q)$ cannot be represented.
- $\mathcal{Y}(P$ **in** $Q) = \mathcal{Y}(P)$ **of** $\mathcal{Y}(Q)$.
- $\mathcal{Y}([s]$ P **in** $Q) = [s]$ $\mathcal{Y}(P)$ **of** $\mathcal{Y}(Q)$.
- $\mathcal{Y}(P$ **with**(k) $Q) = \mathcal{Y}(P)$ **having any** k $\mathcal{Y}(Q)$.
- $\mathcal{Y}([s]$ P **child** $Q) = [s]$ $\mathcal{Y}(P)$ **from** $\mathcal{Y}(Q)$.
- $\mathcal{Y}(P$ **parent**(k) $Q) = \mathcal{Y}(P)$ **intersect** $\mathcal{Y}(Q).parent$. Actually, *parent* cannot be said, but it is necessary to give the constructor name. Thus, this only works if the constructor of Q has always the same constructor as parent and there is no confusion with children of the same constructor.
- **after** and **before** cannot be represented.

B.5 Tree Matching

The definition of the $\mathcal{K}_X$ function follows.

- $\mathcal{K}_X(\textbf{Constr}(c)) = X : c$.
- $\mathcal{K}_X(\textbf{View}(V)) = X : c_1 \ ; \ ... \ ; \ X : c_r$. Note that the language of [KM93] does not have ";" (or), but it says it should, so we use it freely.
- $\mathcal{K}_X(\textbf{Match}(m))$: we disregard this issue here. The results can only be used to restrict other queries.
- $\mathcal{K}_X(P \ \textbf{in} \ Q) = \mathcal{K}_X(P) \ , \ \mathcal{K}_Y(Q) \ , \ (Y : c_1(X) \ ; \ ... \ ; \ Y : c_r(X))$. **beginin** and **endin** are not different from **in** in a single view.
- $\mathcal{K}_X([s] \ P \ \textbf{in/beginin/endin} \ Q)$ cannot be expressed.
- $\mathcal{K}_X(P \ \textbf{with}(k) \ Q) = \mathcal{K}_X(P) \ , \ \mathcal{K}_{Y_1}(Q) \ , \ ... \ , \ \mathcal{K}_{Y_k}(Q) \ , \ (\forall 1 \leq i < j \leq k) \ Y_i \neq Y_j \ , \ (X : c_1(Y_1) \ ; \ ... \ ; \ X : c_r(Y_1)) \ , \ ... \ , \ (X : c_1(Y_k) \ ; \ ... \ ; \ X : c_r(Y_k))$. Note that we need the $\neq$ predicate for $k > 1$. **withbegin** and **withend** are the same as **with** in a single view.
- $\mathcal{K}_X(P \ \textbf{parent/}[s] \ \textbf{child} \ Q)$ cannot be expressed. Only transitive relationships can be seen, this is encouraged in [KM93] to support data independence.
- $\mathcal{K}_X(P \ \textbf{after/before} \ Q)$ cannot be expressed, except what can be done with ordered inclusion in few cases.
- $\mathcal{K}_X(P + Q) = \mathcal{K}_X(P) \ ; \ \mathcal{K}_X(Q)$.
- $\mathcal{K}_X(P - Q)$ cannot be expressed.
- $\mathcal{K}_X(P \ \textbf{is} \ Q) = \mathcal{K}_X(P) \ , \ \mathcal{K}_X(Q)$.
- It is not clear whether **same** can be expressed or not.

It is interesting to observe that $-$, $[s]$ **in**, **child** and **parent** could all be expressible if the language included a closed-world *not* (ala Prolog).

Appendix C

Experimental Results

We present here our experimental results in detail. This appendix is divided into three sections, one for each type of experiment. We begin each section by commenting the specific conclusions derived from the experiments, and then present the tables. More general conclusions are presented in the body of this work.

C.1 An Operator-Wise Test

We present one table for each operator tested. This summarizes fourteen tables, since **before** is tested with $|C| = 0$ and $|C| = 1000$, and $before(k)$ also with $k = 10$ and $k = 500$. For each operator, the five different algorithms are presented, each one for different values for their first and second operand, and for flat and deep operands.

We extract the following specific results from the numbers:

- While the *merge* and *lazy* algorithms are roughly linear, the times for all *search-operate* algorithms grow faster.
- The *search-operate* algorithms are, for sizes exceeding $n = 1000$, uniformly worse than the other versions. They have also higher variance.
- The *merge* and *lazy* algorithms are roughly equivalent, although the *lazy* ones present higher variance. Both versions depend more heavily on the first operand than on the second (except "+", which is symmetric).
- The times for the *merge* versions indicate that approximately 50.000 nodes per second are processed.
- This equivalence between the *merge* and *lazy* versions must be studied for more complex queries, what is done in the next section. We should also notice that some differences are because the *merge* version uses modifying algorithms and the *lazy* version uses generating ones.
- Except for **before** the "12" and "21" versions are not too different, and it is hard to predict what should be applied based on simple measures such as size.
- Almost all times are worse for deep operands than for flat, having larger mean and variance. However, the *merge* algorithms are the less affected.
- When the "12" vs "21" heuristic can be changed for subproblems, the mixed *search-operate* version has larger variance, and can be better or even worse than any of its "12" or "21" counterparts.
- The "12" version for **with** is uniformly worse than its "12" counterpart and therefore should be discarded.

- The *lazy* version of **with** seems to work better for larger values of k. This could indicate that it is more efficient to expand wholly the second operand and compute its sizes than initially avoiding expansion and then having to expand small pieces. This deserves further study.
- In the **before** and **before**(k) algorithms, a larger C affects mean and variance, generally for worse. In a few cases it helps by narrowing the search.
- In the **before**(k) algorithms, k does not affect too much the performance. These algorithms are a little worse than those for **before**.

"+" operator:

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.02	0.14	0.00	0.02	0.15
	10^3	0.02	0.03	0.17	0.02	0.03	0.20
	10^4	0.14	0.17	0.35	0.15	0.20	0.53
lazy	10^2	0.00	0.01	0.24	0.00	0.03	0.23
	10^3	0.01	0.03	0.26	0.03	0.06	0.35
	10^4	0.24	0.26	0.69	0.23	0.35	0.74
search-operate	10^2	0.01	0.01	0.69	0.01	0.02	1.50
	10^3	0.02	0.06	0.72	0.02	0.04	1.52
	10^4	0.69	0.72	1.19	1.50	1.52	2.23
"12"	10^2	0.01	0.01	0.67	0.01	0.02	1.50
	10^3	0.02	0.06	0.71	0.02	0.05	1.51
	10^4	0.69	0.75	1.21	1.50	1.56	2.18
"21"	10^2	0.01	0.02	0.71	0.01	0.03	1.50
	10^3	0.01	0.07	0.75	0.02	0.04	1.58
	10^4	0.69	0.70	1.18	1.49	1.50	2.21

"−" operator:

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.01	0.02	0.14	0.01	0.02	0.16
	10^3	0.02	0.03	0.15	0.02	0.03	0.25
	10^4	0.14	0.15	0.35	0.15	0.18	0.55
lazy	10^2	0.00	0.00	0.03	0.00	0.01	0.05
	10^3	0.01	0.01	0.05	0.03	0.04	0.17
	10^4	0.11	0.12	0.50	0.06	0.11	0.62
search-operate	10^2	0.01	0.02	0.69	0.01	0.02	1.51
	10^3	0.02	0.07	0.70	0.01	0.04	1.53
	10^4	0.65	0.71	1.15	1.47	1.52	1.91
"12"	10^2	0.01	0.02	0.67	0.01	0.01	1.50
	10^3	0.03	0.06	0.72	0.03	0.04	1.52
	10^4	0.76	0.82	1.02	1.55	1.60	2.07
"21"	10^2	0.00	0.02	0.72	0.01	0.02	1.51
	10^3	0.01	0.05	0.75	0.01	0.03	1.53
	10^4	0.64	0.68	1.18	1.49	1.53	2.02

“is” operator:

Version		Flat operands			Deep operands		
	*op*1 \ *op*2	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.02	0.16	0.01	0.02	0.16
	10^3	0.03	0.04	0.17	0.03	0.03	0.24
	10^4	0.26	0.26	0.46	0.28	0.29	0.56
lazy	10^2	0.00	0.00	0.01	0.02	0.04	0.07
	10^3	0.00	0.00	0.03	0.03	0.04	0.14
	10^4	0.03	0.03	0.36	0.03	0.14	0.58
search-operate	10^2	0.00	0.02	0.69	0.01	0.01	1.50
	10^3	0.02	0.06	0.69	0.01	0.04	1.51
	10^4	0.82	0.85	1.01	1.62	1.64	2.06
“12”	10^2	0.01	0.01	0.63	0.01	0.01	1.43
	10^3	0.02	0.06	0.69	0.02	0.04	1.52
	10^4	0.82	0.84	1.04	1.57	1.63	2.36
“21”	10^2	0.00	0.02	0.84	0.01	0.01	1.59
	10^3	0.02	0.05	0.86	0.00	0.02	1.59
	10^4	0.72	0.72	0.94	1.49	1.49	1.93

“in” operator:

Version		Flat operands			Deep operands		
	*op*1 \ *op*2	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.15	0.00	0.02	0.16
	10^3	0.01	0.03	0.16	0.03	0.06	0.18
	10^4	0.15	0.25	0.43	0.28	0.29	0.53
lazy	10^2	0.00	0.00	0.03	0.00	0.02	0.02
	10^3	0.00	0.00	0.04	0.01	0.03	0.04
	10^4	0.12	0.13	0.33	0.04	0.14	0.50
search-operate	10^2	0.01	0.01	0.68	0.00	0.02	1.49
	10^3	0.01	0.05	0.72	0.02	0.03	1.49
	10^4	0.69	0.81	1.17	1.63	1.64	1.93
“12”	10^2	0.01	0.01	0.68	0.00	0.01	1.47
	10^3	0.01	0.04	0.69	0.03	0.03	1.49
	10^4	0.69	0.94	1.20	1.69	1.73	1.93
“21”	10^2	0.01	0.01	0.72	0.00	0.01	1.62
	10^3	0.01	0.06	0.84	0.02	0.02	1.65
	10^4	0.69	0.71	1.15	1.45	1.50	1.84

“[s] in” operator: $s = 1..3$

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.14	0.00	0.02	0.15
	10^3	0.03	0.03	0.16	0.03	0.03	0.20
	10^4	0.22	0.24	0.40	0.23	0.25	0.57
lazy	10^2	0.00	0.00	0.03	0.00	0.01	0.07
	10^3	0.00	0.00	0.04	0.03	0.05	0.16
	10^4	0.07	0.08	0.65	0.12	0.30	0.69
search-operate	10^2	0.00	0.01	0.68	0.00	0.01	1.45
	10^3	0.01	0.05	0.71	0.01	0.03	1.49
	10^4	0.71	0.72	0.92	1.50	1.53	2.13
“12”	10^2	0.00	0.01	0.68	0.00	0.01	1.45
	10^3	0.02	0.06	0.71	0.02	0.02	1.49
	10^4	0.86	0.86	0.92	1.59	1.66	1.89
“21”	10^2	0.00	0.01	0.80	0.00	0.02	1.58
	10^3	0.01	0.05	0.83	0.01	0.05	1.66
	10^4	0.71	0.72	1.18	1.50	1.53	2.13

“with(k)” operator: k=5

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.14	0.00	0.02	0.16
	10^3	0.02	0.03	0.17	0.04	0.05	0.21
	10^4	0.26	0.26	0.45	0.27	0.32	0.66
lazy ($k = 5$)	10^2	0.00	0.01	0.03	0.00	0.02	0.04
	10^3	0.01	0.00	0.04	0.02	0.03	0.13
	10^4	0.04	0.04	0.33	0.02	0.08	0.65
lazy ($k = 100$)	10^2	0.00	0.00	0.01	0.00	0.01	0.05
	10^3	0.00	0.01	0.05	0.00	0.02	0.12
	10^4	0.01	0.03	0.19	0.03	0.05	0.39
search-operate	10^2	0.00	0.01	0.69	0.00	0.04	1.70
	10^3	0.02	0.06	1.07	0.01	0.07	2.34
	10^4	0.74	0.76	1.20	1.53	1.57	2.72
“12”	10^2	0.00	0.01	0.69	0.06	0.07	1.70
	10^3	0.08	0.14	1.07	0.11	0.26	2.34
	10^4	0.97	1.27	1.52	2.44	2.49	2.71
“21”	10^2	0.00	0.02	0.86	0.00	0.02	1.62
	10^3	0.02	0.06	0.88	0.01	0.04	1.69
	10^4	0.74	0.76	1.20	1.53	1.57	2.81

"parent(k)" operator: $k = 1$

Version		Flat operands			Deep operands		
	*op*1 \ *op*2	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.00	0.15	0.00	0.02	0.16
	10^3	0.02	0.04	0.17	0.03	0.03	0.20
	10^4	0.22	0.22	0.48	0.23	0.25	0.60
lazy	10^2	0.00	0.01	0.03	0.00	0.01	0.10
	10^3	0.01	0.01	0.05	0.01	0.03	0.11
	10^4	0.01	0.03	0.62	0.03	0.13	0.68
search-operate	10^2	0.00	0.01	0.69	0.00	0.01	1.47
	10^3	0.01	0.05	0.76	0.01	0.06	1.49
	10^4	0.81	0.86	1.22	1.54	1.76	2.39
"12"	10^2	0.00	0.00	0.69	0.00	0.01	1.49
	10^3	0.01	0.06	0.72	0.02	0.03	1.50
	10^4	0.86	0.86	1.14	1.51	1.83	2.13
"21"	10^2	0.00	0.01	0.82	0.00	0.02	1.51
	10^3	0.01	0.05	0.90	0.02	0.03	1.55
	10^4	0.76	0.77	1.15	1.50	1.53	2.56

"[s] child" operator: $s = 1..3$

Version		Flat operands			Deep operands		
	*op*1 \ *op*2	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.12	0.00	0.02	0.18
	10^3	0.02	0.04	0.17	0.03	0.04	0.20
	10^4	0.24	0.28	0.40	0.28	0.30	0.56
lazy	10^2	0.00	0.00	0.01	0.00	0.00	0.04
	10^3	0.00	0.00	0.03	0.02	0.04	0.16
	10^4	0.03	0.03	0.37	0.05	0.12	0.58
search-operate	10^2	0.00	0.01	0.73	0.00	0.02	1.47
	10^3	0.01	0.06	0.77	0.02	0.04	1.60
	10^4	0.81	0.85	1.08	1.63	1.63	2.31
"12"	10^2	0.00	0.01	0.66	0.01	0.01	1.47
	10^3	0.01	0.05	0.69	0.02	0.04	1.49
	10^4	0.85	0.88	1.04	1.65	1.67	2.43
"21"	10^2	0.00	0.02	0.81	0.00	0.01	1.57
	10^3	0.00	0.05	0.84	0.01	0.03	1.60
	10^4	0.74	0.76	0.98	1.54	1.56	2.03

“before” operator: $|C| = 0$

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.14	0.00	0.03	0.15
	10^3	0.03	0.04	0.17	0.05	0.06	0.21
	10^4	0.31	0.34	0.46	0.34	0.40	0.64
lazy	10^2	0.00	0.00	0.03	0.00	0.02	0.03
	10^3	0.02	0.02	0.05	0.03	0.04	0.05
	10^4	0.28	0.28	0.50	0.30	0.33	0.68
search-operate	10^2	0.00	0.01	0.68	0.00	0.02	0.77
	10^3	0.02	0.06	0.68	0.01	0.05	1.00
	10^4	0.81	0.85	1.79	1.00	1.22	2.38
“12”	10^2	0.00	0.01	0.68	0.00	0.02	0.77
	10^3	0.04	0.07	0.68	0.06	0.07	1.00
	10^4	1.13	1.14	1.79	1.93	1.99	2.38
“21”	10^2	0.00	0.03	0.85	0.00	0.03	1.61
	10^3	0.02	0.06	0.97	0.01	0.05	1.83
	10^4	0.81	0.85	1.44	1.00	1.22	3.89

“before” operator: $|C| = 1000$

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.01	0.03	0.19	0.02	0.04	0.19
	10^3	0.05	0.06	0.43	0.07	0.09	0.46
	10^4	0.35	0.39	0.55	0.39	0.44	0.65
lazy	10^2	0.00	0.01	0.05	0.01	0.03	0.05
	10^3	0.03	0.06	0.06	0.05	0.06	0.08
	10^4	0.34	0.35	0.62	0.35	0.38	0.72
search-operate	10^2	0.04	0.06	0.90	0.04	0.04	0.84
	10^3	0.04	0.11	1.34	0.02	0.11	1.13
	10^4	0.86	1.04	1.97	1.17	1.34	3.76
“12”	10^2	0.04	0.06	0.90	0.05	0.04	0.84
	10^3	0.08	0.08	1.34	0.09	0.15	1.13
	10^4	1.28	1.75	1.97	2.17	2.20	3.76
“21”	10^2	0.04	0.08	0.96	0.04	0.05	1.63
	10^3	0.04	0.12	1.68	0.02	0.11	1.79
	10^4	0.86	1.04	2.68	1.17	1.34	5.05

"before(k)" operator: $k = 10, |C| = 0$

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.00	0.01	0.14	0.00	0.02	0.16
	10^3	0.02	0.03	0.17	0.03	0.07	0.21
	10^4	0.30	0.31	0.42	0.30	0.32	0.62
lazy	10^2	0.00	0.01	0.04	0.00	0.01	0.05
	10^3	0.02	0.02	0.08	0.07	0.09	0.23
	10^4	0.26	0.30	0.55	0.27	0.29	0.63
search-operate	10^2	0.00	0.01	0.62	0.00	0.01	0.68
	10^3	0.01	0.06	0.81	0.01	0.06	1.06
	10^4	0.81	0.84	1.21	0.93	1.14	2.39
"12"	10^2	0.00	0.01	0.62	0.00	0.01	0.68
	10^3	0.03	0.06	0.81	0.03	0.08	1.06
	10^4	1.03	1.11	1.21	1.80	1.81	2.39
"21"	10^2	0.00	0.02	0.84	0.00	0.02	1.63
	10^3	0.01	0.06	0.86	0.01	0.06	1.70
	10^4	0.81	0.84	1.43	0.93	1.14	2.69

"before(k)" operator: $k = 10, |C| = 1000$

Version		Flat operands			Deep operands		
	$op1 \setminus op2$	10^2	10^3	10^4	10^2	10^3	10^4
merge	10^2	0.01	0.02	0.15	0.02	0.02	0.19
	10^3	0.04	0.05	0.20	0.05	0.09	0.22
	10^4	0.26	0.28	0.46	0.32	0.33	0.66
lazy	10^2	0.00	0.00	0.03	0.01	0.03	0.05
	10^3	0.02	0.03	0.08	0.04	0.07	0.21
	10^4	0.34	0.36	0.54	0.34	0.34	0.68
search-operate	10^2	0.04	0.05	0.78	0.03	0.04	0.80
	10^3	0.04	0.07	0.79	0.05	0.11	1.11
	10^4	0.89	0.98	1.45	0.96	1.15	2.42
"12"	10^2	0.04	0.05	0.78	0.04	0.04	0.80
	10^3	0.06	0.07	0.79	0.06	0.10	1.11
	10^4	1.20	1.27	1.45	1.90	1.95	2.42
"21"	10^2	0.04	0.06	1.00	0.03	0.07	1.69
	10^3	0.04	0.07	1.02	0.05	0.11	1.81
	10^4	0.89	0.98	1.76	0.96	1.15	3.11

"before(k)" operator: $k = 500, |C| = 0$

Version	$op1 \setminus op2$	Flat operands 10^2	Flat 10^3	Flat 10^4	Deep operands 10^2	Deep 10^3	Deep 10^4
merge	10^2	0.00	0.01	0.12	0.00	0.02	0.16
	10^3	0.02	0.03	0.16	0.06	0.10	0.18
	10^4	0.26	0.28	0.36	0.33	0.43	0.59
lazy	10^2	0.00	0.01	0.03	0.01	0.03	0.04
	10^3	0.03	0.06	0.13	0.04	0.17	0.27
	10^4	0.32	0.35	0.59	0.35	0.43	0.64
search-operate	10^2	0.00	0.01	0.63	0.00	0.02	0.78
	10^3	0.01	0.07	0.70	0.02	0.06	1.05
	10^4	0.83	0.88	1.15	0.92	1.17	2.29
"12"	10^2	0.00	0.01	0.63	0.00	0.02	0.78
	10^3	0.02	0.07	0.70	0.04	0.04	1.05
	10^4	0.97	1.12	1.15	1.89	1.93	2.29
"21"	10^2	0.00	0.03	0.85	0.00	0.03	1.73
	10^3	0.01	0.07	0.88	0.02	0.06	1.74
	10^4	0.83	0.88	1.20	0.92	1.17	3.05

"before(k)" operator: $k = 500, |C| = 1000$

Version	$op1 \setminus op2$	Flat operands 10^2	Flat 10^3	Flat 10^4	Deep operands 10^2	Deep 10^3	Deep 10^4
merge	10^2	0.00	0.03	0.18	0.01	0.03	0.19
	10^3	0.04	0.05	0.22	0.07	0.11	0.24
	10^4	0.33	0.35	0.48	0.35	0.38	0.62
lazy	10^2	0.00	0.00	0.04	0.01	0.05	0.04
	10^3	0.04	0.07	0.10	0.06	0.09	0.22
	10^4	0.31	0.36	0.62	0.36	0.42	0.71
search-operate	10^2	0.04	0.04	0.81	0.03	0.05	0.81
	10^3	0.04	0.08	0.88	0.03	0.11	1.10
	10^4	0.90	1.04	1.45	0.99	1.23	2.40
"12"	10^2	0.04	0.04	0.81	0.03	0.05	0.81
	10^3	0.06	0.08	0.88	0.06	0.10	1.10
	10^4	1.20	1.27	1.45	1.96	2.15	2.40
"21"	10^2	0.04	0.07	0.99	0.01	0.07	1.84
	10^3	0.04	0.07	1.07	0.03	0.11	1.98
	10^4	0.90	1.04	1.67	0.99	1.23	3.77

C.2 A Test for Lazy vs Merge Algorithms

In this case, we select from a sample file with C code a number of "real-life" queries. For each query we present, we detail the total size of leaves, the total number of nodes from all leaves and intermediate results, the size of the final results, the number of operators in the query, the times taken by the *merge* and *lazy* algorithms to evaluate the query, the percentage *lazy* time / *merge* time, and the percentage of expansion forced by the lazy algorithm, as a fraction of the whole number of nodes that full evaluation forces to compute. To be fair, we do not compute twice repeated operands for full evaluation, since reuse can be accomplished by using generating algorithms.

The specific conclusions we extract are:

- In real cases, *lazy* algorithms seem to be better than *merge* ones, although due to the large variance of the first ones, the fraction is somewhere between 0.25 and 0.90.
- The percentage expanded by lazy evaluation has also large variance, lying somewhere between 40% and 100%, depending on a number of subtle parameters. Some of them are the flatness of the sets, and the relation between initial operand sizes and final result size.
- The total time of evaluation is, in both cases, proportional to the total number of nodes processed, or more roughly, to the sizes of the leaves and the number of operations to perform. That means, for the *merge* version, between $[1.5 .. 2.5] \times 10^{-5}$ seconds per node.
- It seems that, in practice, generating algorithms are better than modifying ones. This is perhaps because final sizes tend to be small in comparison with leaves sizes. This point deserves further study.

Leaves size	Total size	Final size	number of operators	*merge* time	*lazy* time	*lazy/merge* time	*lazy/merge* expansion
1167	1167	1167	0	0.00	0.00	100%	100%
1641	1863	222	1	0.02	0.02	100%	85%
2329	2932	603	1	0.05	0.02	98%	40%
1395	1457	27	2	0.04	0.01	25%	35%
2350	3704	605	2	0.05	0.04	57%	80%
2422	3273	338	2	0.07	0.08	114%	95%
2104	2648	29	3	0.06	0.04	67%	78%
8196	9440	16	3	0.23	0.08	34%	39%
7164	9316	1076	3	0.36	0.14	39%	100%
9117	16832	311	3	0.46	0.20	44%	36%
14174	18260	188	8	0.42	0.33	78%	72%

C.3 A Test for Binary vs Interpolation Search

In this case we took, for sizes 10, 10^2, 10^3 and 10^4, 10 samples of each size. Each sample consisted of two flat operands of that size, which we operated (doing just search) using binary and interpolation search. The result is that interpolation search is normally not advisable under the distributions likely to appear in practice.

Size	Binary	Interpolation
10	0.00	0.01
10^2	0.00	0.01
10^3	0.06	0.09
10^4	1.12	2.87