An Algebra for Structured Text Search

by

Charles L. A. Clarke

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Querying a collection of text may be abstractly viewed as applying a search predicate to all substrings of the text and selecting those substrings that satisfy the predicate as solutions to the query. A single additional restriction is added to this simple model: The substrings accepted as solutions must not properly contain other substrings that satisfy the query. The focus of the thesis is on the properties and applications of this shortest substring search model.

The primary contribution of the thesis is an algebra for structured text search and its implementation framework. The algebra manipulates arbitrary intervals of text, which are recognized in the text from implicit or explicit markup. The algebra has seven basic operators, which combine intervals to yield new ones. The ultimate result of a query is the set of intervals that satisfy it. Implementation is based on four primitive access functions. Recursive definitions for the operators are given in terms of these access functions. A realization of the implementation in terms of inverted lists is examined.

The model is also applied to the general problem of pattern matching in strings, and specifically to the problem of regular expression search. An implementation framework for scanning text with the search algebra is developed. Finally, properties of the model are exploited to develop a new technique for ranking query results according to relevance.

Acknowledgements

Dr. Gordon V. Cormack, my supervisor, introduced me to the area of text databases and encouraged my research into structured text search. During my years as a graduate student, and particularly during the last three years when I was actively involved in the research described in this thesis, he has provided me with endless time, patience, insight, ideas and guidance. He has also become my friend, and to him I shall always be grateful.

Thanks are due to my examining committee: Dr. Ian H. Witten of the University of Waikato, New Zealand; Dr. George H. Freeman of the Electrical and Computer Engineering Department; Dr. Frank W. Tompa and Dr. Kenneth M. Salem. Their comments and suggestions improved the clarity, presentation and correctness of the results. I also thank Dr. T. Bunting of the Geography Department, who acted as chair during the examination.

While I have been graduate student at Waterloo, I have benefited from the ideas and influence of many faculty members and students. I thank in particular: Dr. Forbes J. Burkowski, Dr. Peter A. Buhr, Dr. William B. Cowan, Dave Mason, Rob Good, Elizabeth Tudhope, Gord Vreugdenhil and Crispin Cowan.

The Infomation Technology Research Centre provided financial support for this research through their funding of the MultiText project. The Natural Sciences and Engineering Research Council of Canada provided initial support of my Ph.D. studies through a postgraduate scholarship.

I thank my family for their limitless love and patience.

Charlie Clarke
July 9, 1996, 10:17 p.m.

for
my father

Contents

1 Introduction **1**

2 Models for Structured Text Search **5**

2.1 Document Databases 5

2.2 Models of Text Structure 7

2.2.1 Physical Structure 7

2.2.2 Logical Structure 8

2.2.3 Grammar-Based Hierarchical Model 10

2.2.4 Relational Model 11

2.2.5 A Model for Structured Text Search 12

2.3 Structured Text Search 14

2.3.1 Textriever 16

2.3.2 PAT 18

3 Generalized Concordance Lists **23**

3.1 Definitions and Discussion 23

3.2 Basic Properties 24

4 An Algebra for Structured Text Search **26**

4.1 An Algebra for Structured Text Search 26

4.2 Query Examples 29
4.3 Properties 34
4.4 A Framework for Implementation 44
4.4.1 Index Organization 56
4.5 Correctness of the Framework 61
4.6 Implementation 67
4.6.1 Efficiency 68
4.6.2 Experience 74
4.6.3 Memoization 76
4.7 Generalized Operators 76
4.7.1 Combination 77
4.7.2 Enumeration 79
4.8 Comparison with Other Work 81
4.8.1 Textriever 81
4.8.2 PAT 81
4.8.3 PAT trees vs. Inverted Lists 84

5 Pattern Matching in Structured Text 86

5.1 Background 87
5.1.1 Regular Expressions 87
5.1.2 Pattern Matching with Regular Expressions 88
5.2 Shortest Substrings 90
5.3 Substring Search Algorithms 92
5.3.1 The Complexity of Longest-Match Disjoint Substring Search 92
5.3.2 Shortest-Match Substring Search 93
5.4 Explicit Containment 95
5.5 The `cgrep` Search Tool 97

5.5.1 Basic Searching . . . 98
5.5.2 Search Universes . . . 101
5.5.3 Other Features of `cgrep` . . . 103
5.5.4 Performance . . . 105
5.6 The Structure Algebra Revisited . . . 106

6 Shortest Substring Ranking **110**
6.1 Ranking of Boolean Queries . . . 110
6.2 Ranking by Solution Density . . . 111
6.3 TREC-4 Experiment . . . 112
6.3.1 Overview of TREC . . . 113
6.3.2 Procedure . . . 114
6.3.3 Results and Discussion . . . 117
6.4 Further Related Work . . . 119

7 Conclusion **121**
7.1 Closure . . . 121
7.2 Additional Properties of Generalized Concordance Lists . . . 122
7.3 Future Work . . . 124
7.3.1 Query Plans and Solution Caching . . . 124
7.3.2 Recursive Nesting . . . 124
7.3.3 Indirection . . . 125
7.3.4 Co-Existence with the Relational Model . . . 126
7.3.5 Syntax Error Analysis . . . 126

Bibliography **128**

List of Figures

1.1 Text structure in *Macbeth*. 2

2.1 Tagging the logical structure of *Macbeth*. 9

2.2 A grammar for the structure of a thesis chapter. 10

2.3 A relational table of technical reports. 12

2.4 Indexing for a portion of *Macbeth*. 15

2.5 Operators in Burkowski's algebra. 16

2.6 PAT operators. 20

4.1 Definitions for the binary operators in the query algebra. 28

4.2 Evaluating $A \Diamond B$. 45

4.3 Comparison of the access functions. 47

4.4 τ and ρ for the containment operators. 49

4.5 τ and ρ for the combination operators. 50

4.6 τ and ρ for the ordering operator. 51

4.7 τ' and ρ' for the containment operators. 52

4.8 τ' and ρ' for the combination operators. 53

4.9 τ' and ρ' for the ordering operator. 54

4.10 Evaluating $A \triangleleft B$. 55

4.11 Inverted lists. 58

4.12 Index block organization. 59
4.13 Overview of an update cycle. 60
4.14 Driver procedure. 68
4.15 Architecture of the MultiText System. 75
4.16 Implementation framework for the generalized combination operator. 78
4.17 Implementation framework for enumeration. 80
4.18 Mapping of Burkowski's algebra. 82
4.19 Mapping of PAT operators. 83

5.1 Shortest-match substring search algorithm. 94
5.2 Performance of `cgrep`. 106
5.3 τ for regular expression pattern matching. 108
5.4 ρ for structural pattern matching. 109
5.5 τ' for structural pattern matching. 109

6.1 TREC topic 246. 116
6.2 Query for topic 246. 116
6.3 Comparison of TREC-4 ad-hoc results. 118
6.4 TREC-4 system performance. 120

Chapter 1

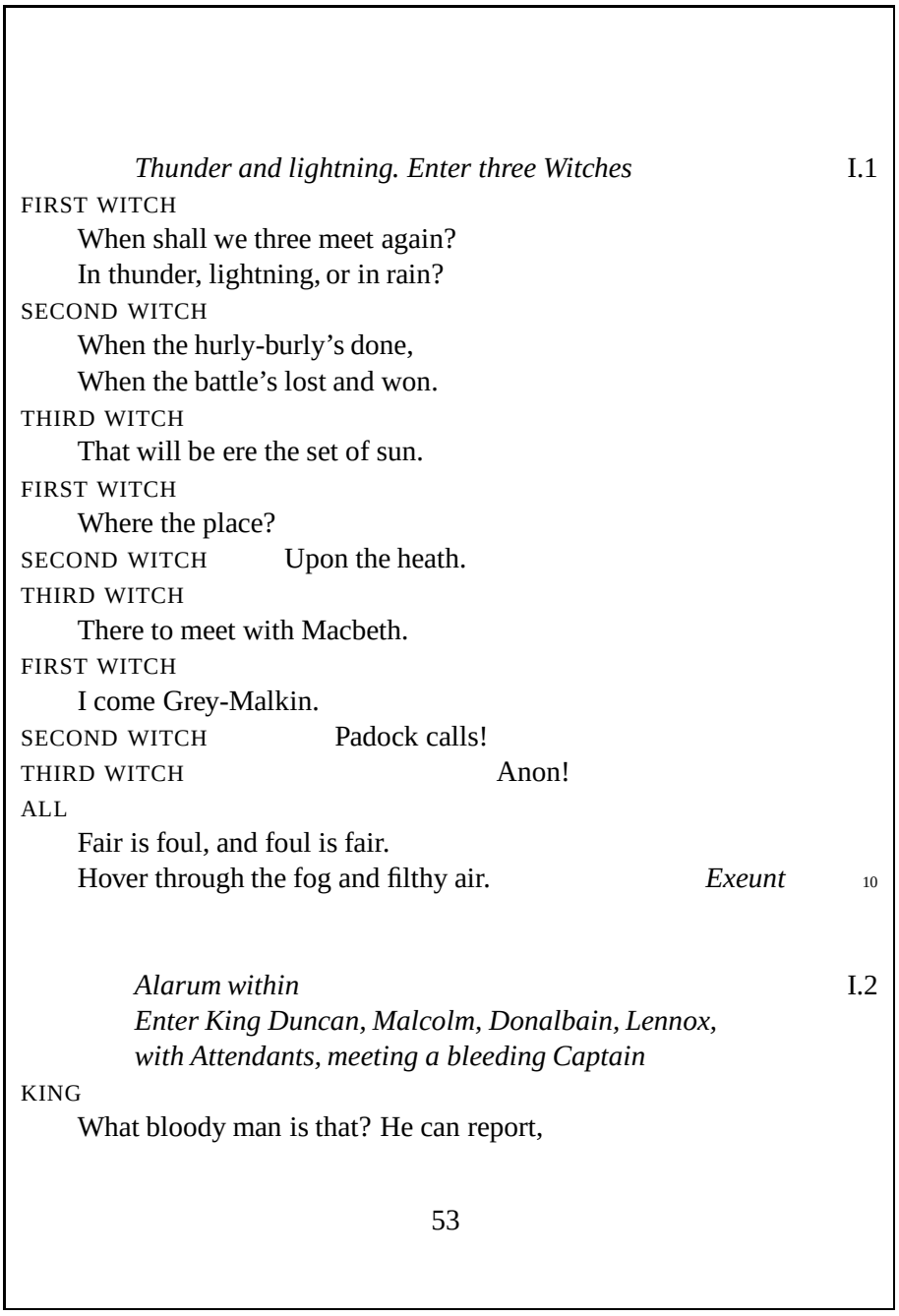
Introduction

Text has natural structure. A document may divide into chapters, pages, sentences, paragraphs, sections, subsections, books, volumes, issues, lines, verses or stanzas. A document may include a title, a preface, an abstract, an epilogue, quotes, references, emphasised passages, digressions and notes. Characteristics of documents vary greatly. A document may have an identified author, or it may be anonymous; it may be precisely dated, or it may be undated; it may be written in Russian using cyrillic characters; it may be written in Japanese using Kanji; it may be part of a larger work; it may stand alone. Each document is structured differently, and the structure may vary even within a document.

Figure 1.1, a facsimile of a page from an edition of Shakespeare's *Tragedie of Macbeth*, demonstrates the complexity of structure in text. This single page includes stage directions, speakers, speeches, the start of the play's first act, its entire first scene and the start of its second. Marginal notations indicate the act, scene and line numbers. Italics, small capitals, and two font sizes are used. The lineation of the spoken verse is consistent with its scansion. A page number appears at the bottom of the page.

This thesis addresses the problem of searching a collection of structured text. Ideally, queries over the collection should be expressed in terms of the natural structure of the text within it. Consider the following queries over a text collection that includes the *Works* of Shakespeare:

1. *Find plays that contain "Birnam" followed by "Dunsinane".*
2. *Find fragments of text that contain "Birnam" and "Dunsinane".*
3. *Find the pages on which the word "Birnam" is spoken by a witch.*



Thunder and lightning. Enter three Witches I.1

FIRST WITCH
When shall we three meet again?
In thunder, lightning, or in rain?
SECOND WITCH
When the hurly-burly's done,
When the battle's lost and won.
THIRD WITCH
That will be ere the set of sun.
FIRST WITCH
Where the place?
SECOND WITCH Upon the heath.
THIRD WITCH
There to meet with Macbeth.
FIRST WITCH
I come Grey-Malkin.
SECOND WITCH Padock calls!
THIRD WITCH Anon!
ALL
Fair is foul, and foul is fair.
Hover through the fog and filthy air. *Exeunt* 10

Alarum within I.2
Enter King Duncan, Malcolm, Donalbain, Lennox,
with Attendants, meeting a bleeding Captain
KING
What bloody man is that? He can report,

Figure 1.1: Text structure in *Macbeth*.

4. *Find speeches that contain "toil" or "trouble" in the first line, and do not contain "burn" or "bubble" in the second line.*

5. *Find a speech by an apparition that contains "fife" and that appears in a scene along with the line "Something wicked this way comes".*

Not only do these examples use document structure to express the query, but the required result — be it play, page, speech, line or merely fragment of text — is also specified in terms of this structure.

Most importantly, text within the collection should not be required to adhere to a particular pre-defined format or schema. Nonetheless, queries should be able to reference structure directly, and to reference equivalent structure across differently formatted documents.

Two themes unify the results of the thesis. The first theme is the structure of text and its effect on text search. The thesis examines the representation of text structure, develops an algebra for expressing structural queries, and discusses the implementation of this algebra, both over indexed text, stored in a text database system, and over un-indexed text, stored in a flat text file. The second theme is a data abstraction, *the generalized concordance list*, that underlies the results. A generalized concordance list is an ordered list of ranges over the sequence of words that make up the text collection. These ranges may overlap, but they may not nest. This seemingly inconsequential property of generalized concordance lists is key to the results.

The plan of the thesis: Chapter 2 first provides the necessary background information concerning the representation of structure in text, making a particular distinction between representation for display and representation for search. The chapter then introduces a simple model for capturing text structure that will be used through much of the rest of the thesis (the main exception being Chapter 5). The chapter concludes with a review of past work in structured text search, with particular emphasis on two text search systems that directly influenced the present work.

Chapter 3 then formally introduces the basic definitions and results associated with generalized concordance lists and the *shortest substring search model*. The chapter is quite short, but as generalized concordance lists are fundamental to the thesis, this material is placed separately.

Chapter 4 presents the core result of the thesis: an algebra for structured text search. The operators of the algebra are defined over generalized concordance lists, and evaluate to generalized concordance lists. The algebra has seven basic binary operators: four *containment operators* that express hierarchical relationships among elements in the text, two *combination operators* that associate elements of the text, and an *ordering operator* that expresses positional relationships among elements of the text. Along with other examples, the algebra is used to express the

Macbeth queries given earlier. Properties of the algebra are then explored. In particular, various commutative, associative and distributive properties are demonstrated. Implementation of the algebra is based on four primitive access functions. Each access function finds the solution to a query that is, in some sense, nearest to a given position in the database. Recursive definitions for the seven operators are given in terms of these access functions. The chapter concludes with two generalizations of operators from the algebra and a comparison with related work.

Chapter 5 applies the shortest substring search model to standard regular expression search over un-indexed text, extending the method to structured text. This new approach to regular expression search has been used as the basis for a searching tool that has proved to be of significant value in a number of applications, including searching folders of email messages and program source code. The chapter also presents an adaptation of the algebra of Chapter 4 to searching un-indexed text.

Finally, Chapter 6 exploits properties of the shortest substring search model to develop a new technique for ranking the results of a text search by relevance to the user.

The structured text model of Chapter 2, the representation of search result as GC-lists (Chapter 3), the algebra of Chapter 4 and its implementation framework have been described in earlier forms by previous publications of the author [27, 28]. The work of Chapter 6 was presented as part of the Fourth Text REtrieval Conference (TREC-4) [29]. All results and techniques described in the thesis have been implemented and tested in practice in the context of the MultiText Project at the Univeristy of Waterloo [77].

Chapter 2

Models for Structured Text Search

A database system that provides storage and search capabilities for a large text collection must represent the structure of this text and permit queries across the structure. At a minimum, the database system might organize the text into documents and provide for search over sets of these documents. If more complex structure must be represented, the database system must model that structure. One readily available source of structural information is the *markup*, or formatting information, embedded in the document for display purposes by a word processing or text processing program. Alternately, we may express the structure in the form of a relation or a hierarchy of structural elements. Once the structure has been modeled, queries over the database must be able to address this structure as part of the search process.

2.1 Document Databases

A simple text database divides the text collection into documents for search and retrieval purposes. Each document is described by a set of *index terms* that are matched against a query posed by a user. The result of a query is the set of documents whose index terms satisfy the query. These documents may then be retrieved and perused by the user, who hopefully finds in them the information she is seeking.

The index terms assigned to a document are usually determined at the time the document is added to the collection. Together, the index terms form the *vocabulary* of the database. The terms may be assigned manually or may be created automatically from the contents of the document itself, by using the words that appear in the document as the basis for the index terms. Usually,

words that are expected to appear in nearly all documents are not used as index terms. For English-language text, this *stop word list* would include such words as "the" and "of", which appear with great frequency in English. Sometimes the words are reduced to their root forms for indexing. For English-language text this *stemming* includes eliminating "-ing" and "-ed" endings. Finally, the list of index terms may be augmented through the use of a thesaurus or similar tools.

Queries are expressed using Boolean operators — AND, OR and NOT — operating over sets of documents. The AND operator intersects two document sets; the OR operator combines two document sets. The NOT operator implements set difference, taking the complement of a document set with respect to a second set, which may be the universe of all documents in the collection. Words act as elementary terms, each representing the set of documents containing that word. For example, the query

"Birnam" AND "Dunsinane"

would evaluate to the set of documents that contain both the words "Birnam" and "Dunsinane". Various extensions may be incorporated into this basic algebra: Word truncation operators select documents containing a word beginning with a specified prefix. Proximity operators select documents on the basis of the distances between word occurrences.

Most commercial text database systems provide an extended Boolean algebra for formulating queries. Salton and McGill [96, chapter 2] review several such systems.

An alternate approach is to treat the user's query as being representative of the type of document the user is seeking, and to compare it statistically with each of the documents in the collection, reporting the best matches. Under this model, it becomes possible to order the documents according to how well they satisfy the user's query under the particular metric in use. These *relevance ranking* techniques have been the subject of considerable study, and are a central concern to researchers in the *Information Retrieval* area [54, 92, 96, 101, 104]. Harman [53] discusses the well-established ranking methods, and provides general historical and background information. Since 1992, Information Retrieval researchers have met annually at the TREC (Text REtrieval Conference) series of conferences to compare relevance ranking techniques through experimentation over a common collection of documents and query topics [52, 54].

One of the simplest and most effective approaches to relevance ranking is the *vector space model*. Under this model, each document and query is treated as a vector in an n-dimensional space, where each dimension represents a word in the vocabulary of the database. Documents and queries may then be compared using vector measures. Weighted measures based on the inner

product of the vectors, comparing the angle between the vectors for ranking, have been found to be particularly effective [93, 95].

Many research and commercial text database systems provide limited support for document structure. Generally, a document is divided into several predefined fields, typically title, author, date, abstract and body. Queries may then refer to these fields. For example, a query might specify that the body of a selected document should contain the word "Birnam" and that the author field should contain "Shakespeare". Large blocks of text are often divided into sentences, paragraphs or other predefined units. Documents may then be selected on the basis of words appearing in the same sentence or paragraph. These techniques for dealing with document structure are inflexible. Document structure that cannot be mapped into predefined fields or textual units is lost and cannot be referenced in a query.

A system for capturing document structure should be flexible enough to accommodate the variations in structure that occur naturally. Unfortunately, this requirement can be at odds with attempts to impose a fixed schema on a database. It should be possible to index all structure in a document thought to be important at the time that the document is added to the database; it should be possible to add further structural indexing at a later time. Furthermore, when a structural element is irrelevant to the document at hand there should be no artificial requirement to index that structural element — it should not be necessary to break a poem into paragraphs.

2.2 Models of Text Structure

2.2.1 Physical Structure

Authors impose structure on text primarily for formatting and display purposes [18]. In a flat text file, newlines, tabs and other whitespace characters are used to dictate layout. Word processing programs, such as WordPerfect or MS-Word [16], use formatting codes embedded in the internal representation of the text to indicate font changes, sectioning, page breaks and other layout and display information. These programs encourage the user to manipulate the text and its structure strictly from a display perspective. Apart from the content of text, the author's only concern is with the visual appearance of the document. At best, the author provides a limited structural description of the document indirectly through the layout information.

Text processing systems, including `troff` [80] and TEX [47, 67, 68] allow the author to manipulate directly the encoded form of a document and provide a limited facility (macros) for

describing abstract document structure. However, the description of structure is completely at the author's discretion. If they wish, authors may describe the structure of their documents purely in terms of layout and format. More importantly, these systems have no facilities for describing document *meta-structure*, the structure of the structure, that a paragraph contains sentences, a play contains lines, and a poem contains verses.

2.2.2 Logical Structure

The markup used by both word processing and text processing systems describes the physical structure of documents when displayed. As an alternative, the markup may describe the logical structure of the document, with the author depending on the document processing software to translate the logical structure into a physical description for display.

Figure 2.1 shows the opening scene of *Macbeth*, with embedded markup describing the logical structure of the text. A tag of the form "<name>" indicates the beginning of a named structural element; a tag of the form "</name>" indicates the end of a named structural element. The logical format includes a specification of act and scene numbers, and delineates the important structural elements: speakers, speeches, lines and stage directions. The tagging does not specify pagination, spacing or font changes. These aspects of presentation are determined when the logical markup is translated to physical markup for display.

If approached systematically, the use of logical markup simplifies the interchange of documents across different systems for processing and display, freeing the document from dependence on particular software for processing and particular physical devices for display [18, 42, 60, 61]. When these documents are stored archivally, this independence can increase the useful life of the documents beyond that of the software which created them. Since the logical structure of different document types will vary, the description of a document in terms of its logical structure depends on the existence of a method for formally specifying the document's meta-structure, and requires that the author adhere to this meta-structure when creating or modifying the document.

The Standard Generalized Markup Language (SGML) [18, 42, 60] is a method for specifying document meta-structure and tagging documents with logical markup. A document in SGML format consists of three components: 1) a *declaration* of the document's lexical characteristics, including the document character set; 2) a *document type definition* (DTD) that describes the document's meta-structure in the form of a grammar; and 3) the text itself, tagged according to SGML conventions. Usually, the first two components will be shared by many documents, and

```
<act> <act-number> 1 </act-number>
<scene> <scene-number> 1 </scene-number>
  <direction> Thunder and lightning. Enter three Witches
  </direction>
  <speaker> First Witch </speaker>
    <speech> <line> When shall we three meet again? </line>
     <line> In thunder, lightning, or in rain? </line> </speech>
  <speaker> Second Witch </speaker>
      <speech> <line> When the hurly-burly's done, </line>
      <line> When the battle's lost and won. </line> </speech>
  <speaker> Third Witch </speaker>
      <speech> <line> That will be ere the set of sun.
      </line> </speech>
  <speaker> First Witch </speaker>
    <speech> <line> Where the place? </speech>
  <speaker> Second Witch </speaker>
      <speech> Upon the heath. </line> </speech>
  <speaker> Third Witch </speaker>
      <speech> <line> There to meet with Macbeth. </line> </speech>
  <speaker> First Witch </speaker>
    <speech> <line> I come Grey-Malkin. </speech>
  <speaker> Second Witch </speaker>
      <speech> Padock calls! </speech>
  <speaker> Third Witch </speaker>
      <speech> Anon! </line> </speech>
  <speaker> All </speaker>
    <speech> <line> Fair is foul, and foul is fair. </line>
    <line> Hover through the fog and filthy air. </line>  </speech>
  <direction> Exeunt </direction>
</scene>
<scene> <scene-number> 2 </scene-number>
  <direction> Alarum within </direction>
  <direction> Enter King Duncan, Malcolm, Donalbain, Lennox with
     Attendants, meeting a bleeding Captain </direction>
  <speaker> King </speaker>
    <speech> <line> What bloody man is that? He can report, </line>
    <line> As seemth by his plight, of the revolt </line>
```

Figure 2.1: Tagging the logical structure of *Macbeth*.

CHAPTER → `<chapter>` TITLE TEXT SECTIONS `</chapter>`
TITLE → `<title>` TEXT `</title>`
SECTIONS → SECTION SECTIONS
SECTIONS → λ
SECTION → `<section>` TITLE TEXT SECTIONS `</section>`
TEXT → **text**
TEXT → λ

Figure 2.2: A grammar for the structure of a thesis chapter.

particular document processing and formatting software may be specialized to specific declarations and DTD's. The text of Figure 2.1 is tagged in a style resembling that of SGML.

It has been widely observed that an SGML DTD may be interpreted as a schema for the purposes of storing SGML documents in a database system [13,14,22,24,27,71,87,88,91]. A query over this database could then reference document structure by referencing the corresponding elements defined in the DTD.

2.2.3 Grammar-Based Hierarchical Model

The structure of text is often modeled as a hierarchy, with meta-structure described using a context-free grammar [12–15,22,24,31,32,46,48,49,60,61,64,70,71,78,85,87,88,91,97,98]. In its essence, a SGML DTD is a context-free grammar (but with recursion limited to a fixed depth specified in the declaration). Figure 2.2 gives a possible grammar for describing the logical structure of a chapter from this thesis. We use a standard notation for specifying the grammar [57, chapter 4], as the notation of SGML is highly specialized and requires considerable explanation. In the grammar, the terminals representing tags are specified literally in `typewriter` font; the symbol λ is used to represent the empty string; the only other terminal is for **text**, the format of which is not specified; non-terminals are specified in SMALL CAPITALS using descriptive names. Note that any SECTION may have several (sub)SECTIONS contained within it. Instead of allowing arbitrary nesting of sections, the recursion could be explicitly limited to SUBSUBSECTIONS, which would suffice for the present thesis.

Gonnet and Tompa [46] describe an algebra of operations for manipulating a text database that has been parsed according to a context-free grammar. Colby and Van Gucht [31] build on the grammar-based model of Gonnet and Tompa, providing grammar templates for a number of

traditional data models and extending the grammar-based model to support hypertext. Gyssens et al. [49] take a similar approach, presenting two languages for transforming parsed text and showing the languages to be equivalent. The grammar-based model has been proven in practice for restructuring a large body of highly-structured text [15].

Document structure is not always strictly hierarchical. Consider the relationship between speeches and lines in the example of Figure 1.1 and Figure 2.1. The play is broken into lines to reflect its rhythm and verse. Each speech consists of the words spoken as a unit by a single speaker or group of speakers. A speech may include several lines, or a line may be divided across several speeches. The divisions of the play into speeches and into lines are only weakly related, but both are clearly part of the play's logical structure.

There is a similar independence between the physical structure and the logical structure of a document, and in some cases both must be preserved in a database. For example, page numbers are essential in citing U.S. federal court decisions in U.S. federal courts. Ownership of these page numbers for citation purposes has been enforced by the major publisher of U.S. legal decisions [105]. Databases not licensed by this publisher cannot index and report citations using the copyrighted page numbers, making an unlicensed database effectively worthless.

The limitations of the hierarchical model are explicitly recognized in SGML. The CONCUR feature allows secondary DTD's to be associated with a document. Tags defined by the various DTD's may be arbitrarily interleaved in the document, with the tags of the secondary DTD's explicitly labeled to indicate the DTD from which they are drawn. A primary purpose of the CONCUR feature is to allow logical and physical markup to be stored together in the same document [18].

2.2.4 Relational Model

The success of the relational model [30, 36] suggests its extension to structured text [7, 13, 14, 72, 88]. A group of documents might be organized into a relation, with each tuple corresponding to a document and each attribute corresponding to a structural element. The example in Figure 2.3 shows a table of technical reports. Columns of the table correspond to titles, authors, abstracts and other document elements. Each row of the table provides information for a single technical report. The example shows the specific information for Reference 26. This simple mapping of documents into relations is essentially the approach taken by the proposed SFQL standard [7].

The relational model may be combined with the hierarchical model. The Atlas text database system supports document structure using a nested relational model [88]. Nested relations allow

Reports:

title	authors	number	date	abstract	body
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
Fast Inverted Indexes with On-line Update	C. L. A. Clarke G. V. Cormack F. J. Burkowski	CS-94-40	Nov. 23, 1994	We describe data structures and an update strategy for the...	Our general concern is the construction of a distributed full-text information...
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Figure 2.3: A relational table of technical reports.

hierarchical structure to be directly supported. Blake et al. [13,14] add a TEXT data type to SQL, with the structure of a TEXT field defined using an SGML grammar. This approach has since been proposed for standardization [37].

2.2.5 A Model for Structured Text Search

The models for document structure discussed in this section all place significant constraints on the type of structure that can be represented. The model advocated by this thesis records text structure without placing constraints upon it. Constraints and schema considerations are deferred until query formulation.

A text database is represented as a string of concatenated symbols $a_1 \ldots a_N$ drawn from a *text alphabet* Σ_T and a *stoplist alphabet* Σ_S, where $\Sigma_T \cap \Sigma_S = \emptyset$. An index function $\mathcal{I}_T$ maps each symbol in the text alphabet to the set of positions in the database string where the symbol appears ($\mathcal{I}_T : \Sigma_T \rightarrow 2^{\{1 \ldots N\}}$). No equivalent index function for symbols from the stoplist alphabet is defined; symbols from the stoplist alphabet serve merely to occupy positions in the database string and to maintain proximity relationships. The text alphabet and the stoplist alphabet are together referred to as the *database alphabet* $\Sigma_D (= \Sigma_T \cup \Sigma_S)$. The document is marked up using symbols drawn from a *markup alphabet* Σ_M, where $\Sigma_D \cap \Sigma_M = \emptyset$. An index function $\mathcal{I}_M : \Sigma_M \rightarrow 2^{\mathbb{Q}}$ maps each symbol in the markup alphabet to a set of rational numbers corresponding to the associated positions in the database string. Symbols in the markup alphabet do not appear in the database string; they are used only for indexing purposes. Combining the text alphabet and the markup alphabet into a single *index alphabet* $\Sigma = \Sigma_T \cup \Sigma_M$, we define the index function $\mathcal{I} : \Sigma \rightarrow 2^{\mathbb{Q}}$ as the union of $\mathcal{I}_T$ and $\mathcal{I}_M$. For convenience we define the value ϵ to be

the smallest positioning quantum in the database — ϵ is the largest rational number for which p/ϵ is an integer for every value p in the range of the index function $\mathcal{I}$. It is important to emphasize that ϵ is an attribute of each particular database, and will vary from database to database.

A reasonable representation of the document in Figure 1.1 in our text database model uses words as the text alphabet:

$$\Sigma_T = \{ \texttt{again}, \texttt{air}, \texttt{alarum}, \texttt{all}, \texttt{anon}, \texttt{attendants}, \texttt{battle}, \texttt{be}, \texttt{bleeding}, \texttt{bloody}, \texttt{burly}, \texttt{calls}, \texttt{can}, \texttt{captain}, \texttt{come}, \ldots \}.$$

The stoplist alphabet consists of words that occur most commonly in English text:

$$\Sigma_S = \{ \texttt{and}, \texttt{for}, \texttt{in}, \texttt{is}, \texttt{of}, \texttt{that}, \texttt{the}, \texttt{to}, \texttt{said} \}.$$

In this instance we made the arbitrary choice to ignore case and punctuation in creating the database alphabet. The symbols from the database alphabet are concatenated in the order they appear textually to form the database string:

```
thunder and lightning enter three witches first witch when
shall we three meet again in thunder lightning or in rain
second …
```

The markup tags of Figure 2.1 become the basis for the markup alphabet, to which we might add indexing to describe the physical as well as the logical structure. A possible indexing for a portion of our example document is given in Figure 2.4. In the figure, markup symbols are generally indexed at integer positions. It is only in the cases that a structural element begins and ends at the same word that we index a markup symbol halfway-between two database symbols (and so $\epsilon = \frac{1}{2}$ in this case). In those cases, occurring at positions 73 and 74, the start tag is indexed at the previous half-word position. This indexing serves to preserve the ordering relationship between start and end tags, a property which will prove useful during query formulation. No symbols are indexed at positions 76 and 78 since the word “is” appears in the stoplist alphabet.

The details of this indexing are arbitrary, and there are reasonable alternatives to it. We could choose to index all markup symbols at the halfway point between database symbols, or choose to order the markup symbols between database symbols and give each a unique position. The exact choice depends on details of implementation and loading. In the experience of the author it is usually best to limit positions to rational numbers where the denominator is a small fixed power

of 2, in order to simplify the physical representation of the positions. It is then necessary to store only the numerators of these rational numbers in the database.

2.3 Structured Text Search

Many proposals for structural text search are direct generalizations of the document database approach described in Section 2.1. The generalization is two-fold:

1. Extension of set-oriented Boolean search and relevance ranking to document components:

   find sections containing the words "information" and "retrieval"

2. The introduction of queries relating the structural elements themselves:

   find titles of sections containing the word "database"

Sacks-Davis et al. [87] discuss this generalization in the context of a database system for SGML documents.

When document structure is modeled using a meta-structural description, by a grammar or a relation, query formulation is usually heavily dependent on details of the meta-structure. The query language developed for the MULTOS project [12] allows documents to be selected on the basis of the contents of specific components, under a hierarchical data model. Macleod [71] describes a language for selecting document components from an SGML database according to Boolean search criteria and structural containment relationships. In that language, elements of solution sets must be of the same component type, with the possible types defined by the non-terminals of the structural grammar. A similar language described by Navarro and Baeza-Yates [78] adds support for ordering relationships between components and provides for multiple hierarchies defined by separate grammars. In that language, a solution set may be composed of components of different types, provided that the components are taken from the same hierarchy.

Kilpeläinen and Mannila [64] describe a tree-based pattern matching language for querying text structured according to a context-free grammar. Christophides et al. [24] describe a mapping of SGML documents into an object-oriented data model and describe extensions to an objected-oriented database system for querying a collection of these documents. Navarro and Baeza-Yates [78] provide an efficiency and expressiveness comparison of the various approaches, including in the comparison a previously-published version of the algebra developed in Chapter 4 [27].

61	62	63	64
first	witch	i	come
<speaker>	</speaker>	<speech>	
		<line>	
65	66	67	68
grey	malkin	second	witch
	</speech>	<speaker>	</speaker>
69	70	71	72
paddock	calls	third	witch
<speech>	</speech>	<speaker>	</speaker>
$72\frac{1}{2}$	73	$73\frac{1}{2}$	74
	anon		all
<speech>	</speech>	<speaker>	</speaker>
	</line>	<line>	
75	76	77	78
fair		foul	
<speech>			

Figure 2.4: Indexing for a portion of *Macbeth*.

Operator	*Meaning*
$A \overset{\uparrow}{\cap} B$	Intersect high
$A \overset{\downarrow}{\cap} B$	Intersect low
$A \overset{\uparrow}{-} B$	Difference high
$A \overset{\downarrow}{-} B$	Difference low
$A \overset{\uparrow}{\cup} B$	Union high
$A \overset{\downarrow}{\cup} B$	Union low
$n \triangleright A$	Extension right
$n \triangleleft A$	Extension left

Figure 2.5: Operators in Burkowski's algebra.

We conclude the chapter with a detailed examination of two systems for structured text search that express the results of search in a manner independent of document meta-structure. In both cases, query results are represented as sets of intervals over the underlying text.

2.3.1 Textriever

Burkowski [21, 22] describes a language for querying hierarchically ordered text databases. The language was developed and implemented as a part of the Textriever system created by SigScan Systems of Toronto.

The language assumes that the text is marked up with a fixed schema in the style of SGML. As in the model described in Section 2.2.5, text in the collection is treated as a single sequence of words, with each word indexed at an integral position. Structure is represented by statically defined lists of word position pairs called *concordance lists*. Members of a concordance list are not permitted to partially overlap one another, or to nest, one within the other. A concordance list describing each structural element is created from the schema and markup at the time a document is scanned and parsed for loading into the database. The index list for a particular word is considered to be a concordance list in which each element is a pair with equal end points, each describing the position of a single occurrence of the word.

The primary focus of the language is on the selection of elements from concordance lists based on containment relationships. The operators in the algebra are given in the table of Figure 2.5.

The *intersect high* operator ($A \Cap\!\!\uparrow B$) selects elements from its left-hand operand (the concordance list A) that contain an element from its right-hand operand. The *intersect low* operator ($A \Cap\!\!\downarrow B$) selects elements from its left-hand operand (A) that are contained in an element from its right-hand operand. The *difference high* ($A \overset{\uparrow}{-} B$) and *difference low* ($A \overset{\downarrow}{-} B$) operators express negated versions of the intersect operators. The difference high operator selects elements from its left-hand operand that do not contain an element from its right-hand operand. The difference low operator selects elements from its left-hand operand that are not contained in an element from its right-hand operand. Note that the result of an intersect or difference operation is automatically a concordance list, since the result is always a subset of the left operand.

There are two union operators that combine concordance lists. The result of a *union high* ($A \Cup\!\!\uparrow B$) operator is the union of its operands with the exception that if one element of the result is nested in another, only the outermost is retained. The result of a *union low* ($A \Cup\!\!\downarrow B$) operator is the union of its operands with the exception that if one element of the result is nested in another, only the innermost is retained.

The algebra further contains two extension operators that may be used to express adjacency and proximity relationships. These extension operators increase the bounds of the elements of a concordance list by a specified integral value. The *extension right* operator ($n \triangleright A$) extends each element of A right by n word positions; the *extension left* operator ($n \triangleleft A$) extends each element of A left by n word positions. Taking an example from Burkowski [22, page 342] the expression

$$(3 \triangleright \text{“foul”}) \Cap\!\!\uparrow \text{“fair”}$$

locates text in which “fair” follows within three words of “foul”.

Unfortunately, the extension operators create a semantic problem within the algebra. Consider the source text

> Spam, spam, spam, spam!
> Spam, spam, spam, spam!

The expression

$$1 \triangleright \text{“spam”}$$

creates a series of overlapping intervals which cannot be expressed as a concordance list. This problem is neither recognized nor addressed by Burkowski.

Similarly, the union operator requires that the concordance lists be “hierarchically compatible”, with no element of one operand partially overlapping an element of the other. Otherwise,

the result of the union operation would not be a concordance list.

A possible solution to these problems is to relax the condition that elements of a concordance list cannot overlap. An approach to relaxing this condition will be the subject of Chapter 3.

2.3.2 PAT

The PAT[1] text searching system [39, 43, 90] was originally developed at the Centre for the New Oxford English Dictionary at the University of Waterloo. The system has since been commercialized by Open Text Corporation.

Under the PAT system, the source text is treated as a string of characters, with indexing specifying character positions within this string. Although indexing specifies character positions, generally only the start positions of words and the locations of markup tags are actually indexed. Markup is treated as a part of the text; the only distinction made between tags and text is for the purposes of indexing.

A PAT operation results in a set with one of two data types: a *match point set* or a *region set*. Each element of a match point set, a match point, is a single character position in the source text. A region specifies a range of text as a pair of character positions, a start position and an end position. A region set enforces the main property of a concordance list, the members of a region set may not overlap or nest.

While both match point sets and region sets are concordance lists, the distinction between them is of considerable importance in PAT. A key aspect of PAT is the use of implicit conversions from region sets to match point sets. The conversion is achieved by discarding the end position from each region in a region set. This conversion is a normal part of many PAT operations, and it is also used to help address the semantic problems associated with taking a union of two concordance lists (region sets) that are not hierarchically compatible.

The most basic search in PAT is for a phrase (or a lexicographically ordered range of phrases). A phrase search results in a match point set specifying the start position of matching phrases in the source text. For example, the result of the query

```
"to be or not to be"
```

[1] PAT is a registered trademark of Open Text Corporation.

is the character positions where an occurrence of the phrase begins. Literal positions in the text may be specified with an expression of the form

`[`n`]`

where n is a positive integer. The result of such an expression is a match point set with a single element. Literal values may be added to each element of a match point set using an expression of the form

`shift.`n m

where n is an integer and m specifies a match point set. If m specifies a region set an implicit conversion is performed.

The remaining PAT operators are summarized in the table of Figure 2.6. The first nine operators are related to structured text search. The last two operators, "`signif`" and "`lref`", are used to find frequently occurring and repeated substrings of the text, and are not related to the structured search aspect of PAT.

A region definition uses two match point sets to specify the start and end positions for regions in a region set. For example, the region definition

```
plays = "<play>"..(shift.6 "</play>")
```

defines a region set for plays, naming it "`plays`". In this definition, the "`shift`" operator is used to move the match point set for the end tag ("`</play>`") six characters to the right. Without the "`shift`" operation, the regions in the set would cover only the "`<`" character of their end tags.

Since elements of a region set cannot overlap, region definitions that would result in a set with self-overlapping regions must be resolved into a region set with the non-overlapping property. Salminen and Tompa [90] give the example of a headline

```
<h>Editor denies <h>Massive Failure</h> misleading</h>
```

and state that the region definition

```
headline = docs "<h>"..(shift.3 "</h>")
```

would "include the region corresponding to the substring

Operator	*Meaning*
`docs` m_0`..`m_1	Region definition
e `including.`n m	Elements of e containing at least n match points from m
e `not including.`n m	Elements of e containing fewer than n match points from m
e `^` m	Elements of e starting at a match point in m
e `-` m	Elements of e not starting at a match point in m
e `fby.`n m	Elements of e starting no more than n characters before a match point in m
e `near.`n m	Elements of e starting no more than n characters from a match point in m
e `within` r	Elements of e contained in a region from r
e_0 `+` e_1	Set union
`signif.`n m	Most frequent repetitions of text
`lref.`n m	Longest repetitions of text

Figure 2.6: PAT operators.

```
<h>Massive Failure</h>
```

but not the surrounding headline." The fragments

```
<h>Editor denies <h>Massive Failure</h>
<h>Massive Failure</h> misleading</h>
```

are also rejected from the region set. Presumably embedded match points from either match point set are not permitted. Further, consider the region definition

```
hh = docs "<h>".."<h>"
```

All but the last of the potential members of this region set will overlap the following member by a character. This problem is not addressed by Salminen and Tompa.

For the operators of Figure 2.6, when a match point set is required as an operand and a region set is provided as an actual operand an automatic type conversion is performed. This behavior produces effects which may be unexpected. Consider another example from Salminen and Tompa

```
line = docs "<line>"..(shift.6 "</line>")
speech = docs "<speech>"..(shift.8 "</speech>")
*line including *speech
```

This search does not produce all lines containing a complete speech, as might be expected, but rather all lines containing the start of a speech. Salminen and Tompa point out that checking the start position is sufficient to confirm the containment of an entire region when the regions are nested. However, speeches are not strictly nested in lines. A line might contain complete speeches, a speech might contain complete lines, a line might be broken across speeches, or a speech might be broken across lines. The single page from *Macbeth* in Figure 1.1 contains examples of all these structural relationships.

If both operands of a union operation ("+") are match point sets, or if both operands are non-overlapping region sets, the result of the operation is the normal set union of the two operands. If one operand is a match point set and the other is a region set, the region set is first converted to a match point set before the operation is performed. If the operands are region sets with overlapping regions they are both converted to match point sets before the operation is performed.

Under these rules, the data type of a union of two region sets is entirely data dependent. Minor additions or deletions of material can easily have significant effect on the results of a query.

Salminen and Tompa give the following example from a database containing the *Complete Works of Shakespeare*:

The query

```
(*speech including "to be or not to be") + (*line including "dying")
```

results in a region set with 58 elements, while the query

```
(*speech including "to be or not to be") + (*line including "death")
```

results in a match point set with 1030 elements. The difference in data type is due to two occurrences of the word "death" in Hamlet's soliloquy. Concerning this example, Salminen and Tompa state:

> Although these semantics are self-consistent, this feature of PAT is easily misunderstood and therefore changing it should be considered, perhaps by suitably defining the union of overlapping regions (as done for example, in [Burkowski [21]]).

Chapter 3

Generalized Concordance Lists

This short chapter introduces *generalized concordance lists*, the abstract data type or mathematical object on which the results of the thesis will find their foundation. After giving basic definitions, we prove a number of simple, fundamental results concerning generalized concordance lists for use in later chapters.

3.1 Definitions and Discussion

The result of a search is represented as a set of ranges or *extents* over the database string. Each extent is of the form (p, q), where $p \in \mathbb{Q}$ is the start position of the extent and $q \in \mathbb{Q}$ is the end position of the extent. The length of an extent is $q - p + \epsilon$, making the smallest extents of length ϵ. An extent (p, q) *overlaps* an extent (p', q') if either $p' \leq p \leq q'$ or $p' \leq q \leq q'$ but not both. An extent (p, q) is *nested* in an extent (p', q') if $(p, q) \neq (p', q')$ and $p' \leq p \leq q \leq q'$. If $a = (p, q)$ and $b = (p', q')$ are extents, the notation $a \sqsubset b$ indicates that a nests in b; the notation $a \sqsubseteq b$ indicates that a is *contained in* b: that either a and b are equal or that a nests in b. Extents form a partial order under $\sqsubseteq$.

Over the database string $a_1..a_N$, the range of the index function $\mathcal{I}$ is limited to $M = \epsilon N$ positions. A search over the database may be satisfied by $O(M^2)$ extents. If the query is for a particular word, every extent that includes an occurrence of the word may be considered a solution to the query. A search to find a word that occurs exactly once in the database is satisfied by at least M extents and by as many as $(\lceil M/2 \rceil + 1)(\lfloor M/2 \rfloor + 1)$, depending on the position of the word in the database string. However, many of these extents overlap and nest. In order

to reduce the number of extents that result from a search, extents that have other result extents nested in them are eliminated from the result set. This approach to eliminating nested extents is termed the *shortest substring search model*.

A set of non-nested extents is referred to as a *generalized concordance list*, or simply *GC-list*, after the *concordance lists* of Burkowski [22]. Burkowski's concordance lists have the property that the element extents must be non-overlapping as well as non-nesting, forcing an arbitrary selection of one extent in favor of another. In the case of a search for a single word that occurs once in the database, the resultant generalized concordance list contains a single extent of length ϵ that begins and ends at the word's position.

The index function $\mathcal{I}$ may be viewed as mapping symbols in the index alphabet onto GC-lists: The elements of the results are interpreted as extents that begin and end at a single position.

We formalize the reduction of a set of extents to a generalized concordance list as a function $\mathcal{G}$ (S):

$$\mathcal{G}(S) = \{a \mid a \in S \text{ and } \nexists\, b \in S \text{ such that } b \sqsubset a\}$$

A set S is generalized concordance list if and only if

$$S = \mathcal{G}(S)$$

3.2 Basic Properties

Theorem 3.1
The elements of a GC-list are totally ordered by their start and end positions. The total orders are identical.

Proof:

Let S be a GC-list. If $a = (p, q) \in S$ and $b = (p', q') \in S$ then either 1) $p < p'$ and $q < q'$ (and therefore $a < b$), 2) $p > p'$ and $q > q'$ (and therefore $a > b$), or 3) $p = p'$ and $q = q'$ (and therefore $a = b$). Otherwise, if $p \geq p'$ and $q < q'$, or if $p > p'$ and $q = q'$, then $a \sqsubset b$, and S would not be a GC-list; if $p' \geq p$ and $q' < q$, or if $p' > p$ and $q' = q$, then $b \sqsubset a$, and S would not be a GC-list. □

Theorem 3.2
If a GC-list S represents the solution to a query over the database string $a_1 ... a_N$ then $|S| \leq \epsilon N$.

Proof:

If $|S| > \epsilon N$ then two distinct elements of S, (u, v) and (u, w), must share a common start position. Since these elements are distinct, $v \neq w$. If $v < w$ then $(u, v) \sqsubset (u, w)$. If $w < v$ then $(u, w) \sqsubset (u, v)$. In either case S cannot be a GC-list. □

Theorem 3.3
If A is a finite set of extents with $a \in A$ and $a \notin \mathcal{G}(A)$ then there exists $a' \in \mathcal{G}(A)$ such that $a' \sqsubset a$.

Proof:

From the definition of GC-lists, there exists at least one extent of A that nests in a. Let n be the number of extents $(a_1, a_2, ..., a_n)$ such that $a_i \sqsubset a$, $\forall 1 \leq i \leq$n. We now proceed by induction on n. For the basis case, if $n = 1$ then there cannot exist $a'' \in A$ such that $a'' \sqsubset a_1$, for then $a'' \sqsubset a$. Thus $a_1 \in \mathcal{G}(A)$. Now consider $n > 1$, and assume the theorem true for $n - 1$ and smaller. If $a_1 \in \mathcal{G}(A)$ then we are done. Otherwise, since a_1 cannot contain more than $n - 1$ elements, by the induction hypothesis there exists $a' \in \mathcal{G}(A)$ such that $a' \sqsubset a_1 \sqsubset a$. □

The theorem does not hold for a non-finite set of extents. Consider the set of extents

$$A = \{(-1/n, 1/n) \mid \forall n,\ n \text{ a natural number}\}$$

Assume there is an element of A, $a = (-\frac{1}{i}, \frac{1}{i})$, such that $a \in \mathcal{G}(A)$. From the definition of A, the extent $a' = (-\frac{1}{i+1}, \frac{1}{i+1})$ is also an element of A. Since $a' \sqsubset a$, a' cannot be an element of $\mathcal{G}(A)$, a contradiction. Therefore, $\mathcal{G}(A) = \emptyset$. For the remainder of the thesis we deal only with finite sets.

Chapter 4

An Algebra for Structured Text Search

In this chapter we develop an algebra for structured text search that uses GC-lists as its only data type. Each operator in the algebra operates uniformly over GC-lists, taking GC-lists as operands and producing GC-lists as results.

After defining the operators, we provide examples of their use and present some of their properties. A framework for implementing the algebra is described and proven correct. The efficiency of an implementation based on this framework is then examined. Finally, we describe two simple generalizations of the operators.

4.1 An Algebra for Structured Text Search

The query algebra has seven binary operators. The operators are presented in Figure 4.1. Each operator is defined over GC-lists and evaluates to a GC-list. The operators fall into three classes: containment, combination and ordering.

The *containment operators* select the elements of a GC-list that are contained in, not contained in, contain, or do not contain the elements of a second GC-list. The containment operators are used to formulate queries that refer to the hierarchical characteristics of structural elements in the database. The expression on the right-hand side of a containment operator acts as a filter to restrict the expression on the left-hand side — the result of the operation is a subset of the result of the left-hand side.

The two *combination operators* are similar to the standard Boolean operators AND and OR. The "both of" operator is similar to AND: Each extent in the result contains an extent from each

operand. The "one of" operator merges two GC-lists: Each extent in the result is an extent from one of the operands.

The *ordering operator* generalizes concatenation. Each extent in the result starts with an extent from the first operand and ends with an extent from the second operand. The extent from the first operand will end before the starting position of the extent from the second operand. The ordering operator may be used, for example, to connect markup alphabet symbols that delineate structural elements, producing GC-lists in which each extent corresponds to one occurrence of the structural element. Examples showing the use of all of these operators are given in the next section.

In addition to the seven binary operators, we define two unary *projection operators*, π_1 and π_2, over both extents and GC-lists. For extents we define:

$$\pi_1(p,q) = (p,p)$$

$$\pi_2(p,q) = (q,q)$$

and if A is a GC-list we define:

$$\pi_1(A) = \{(p,p) \mid \exists q \text{ with } (p,q) \in A\}$$

$$\pi_2(A) = \{(q,q) \mid \exists p \text{ with } (p,q) \in A\}$$

We may define GC-lists containing all extents of a fixed size. Remembering (from Chapter 2) that the value ϵ is defined to be the smallest positioning quantum for a particular database string, we define Σ^x to represent the GC-list of all extents of length x:

$$\Sigma^x = \{(p,q) \mid q - p + \epsilon = x\}$$

An extent of smallest size, having $p = q$, is defined to have length ϵ.

As previously described, the index function $\mathcal{I}$ may be viewed as mapping symbols in the index alphabet onto GC-lists where the start and end points of the elements are equal.

Containment Operators

Contained In:
$$A \lhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } a \sqsubseteq b\}$$

Containing:
$$A \rhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } b \sqsubseteq a\}$$

Not Contained In:
$$A \not\lhd B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } a \sqsubseteq b\}$$

Not Containing:
$$A \not\rhd B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } b \sqsubseteq a\}$$

Combination Operators

Both Of:
$$A \triangle B = \mathcal{G}\left(\{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ and } \exists\, b \in B \text{ such that } b \sqsubseteq c\}\right)$$

One Of:
$$A \bigtriangledown B = \mathcal{G}\left(\{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ or } \exists\, b \in B \text{ such that } b \sqsubseteq c\}\right)$$

Ordering Operator

Before:
$$A \lozenge B = \mathcal{G}\left(\{c \mid \exists\, (p,q) \in A \text{ and } \exists\, (p',q') \in B \text{ where } q < p' \text{ and } (p,q') \sqsubseteq c\}\right)$$

Figure 4.1: Definitions for the binary operators in the query algebra.

4.2 Query Examples

Our algebra may be used to express the *Macbeth* queries given in Chapter 1.

1. *Find plays that contain "Birnam" followed by "Dunsinane".*

$$(\mathcal{I}(\text{“<play>”}) \lozenge \mathcal{I}(\text{“</play>”})) \rhd (\mathcal{I}(\text{“birnam”}) \lozenge \mathcal{I}(\text{“dunsinane”}))$$

   The ordering operation is used to build a GC-list of plays and a GC-list of text fragments that begin with "Birnam" and end with "Dunsinane". Each extent in the result of the expression $\mathcal{I}(\text{“<play>”}) \lozenge \mathcal{I}(\text{“</play>”})$ exactly delimits the extent of a play. The GC-list of text fragments that begin with "Birnam" and end with "Dunsinane" is used to select from the GC-list of plays.

2. *Find fragments of text that contain "Birnam" and "Dunsinane".*

$$\mathcal{I}(\text{“birnam”}) \triangle \mathcal{I}(\text{“dunsinane”})$$

   Since no ordering is specified, the "both of" operator is used. Member extents of the resulting GC-list either begin with "Birnam" and end with "Dunsinane", or begin with "Dunsinane" and end with "Birnam".

3. *Find the pages on which the word "Birnam" is spoken by a witch.*

$$\text{PAGES} \rhd (\mathcal{I}(\text{“birnam”}) \lhd (\text{BIRNAM} \lhd (\text{S} \rhd \text{WITCH})))$$

   where

$$\begin{aligned}
\text{PAGES} &\equiv \mathcal{I}(\text{“<page>”}) \lozenge \mathcal{I}(\text{“</page>”}) \\
\text{BIRNAM} &\equiv (\mathcal{I}(\text{“<speech>”}) \lozenge \mathcal{I}(\text{“</speech>”})) \rhd \mathcal{I}(\text{“birnam”}) \\
\text{S} &\equiv \mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speech>”}) \\
\text{WITCH} &\equiv (\mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speaker>”})) \rhd \mathcal{I}(\text{“witch”})
\end{aligned}$$

   The expressions WITCH and BIRNAM specify speakers that are witches and speeches that contain "Birnam" respectively. The expression S links speaker and speech together. The query is arranged to use the actual occurrence of the word "Birnam" to select pages. If a speech by a witch stretched between two pages and contained an occurrence of "Birnam" on each page, both pages would be selected.

4. *Find speeches that contain "toil" or "trouble" in the first line, and do not contain "burn" or "bubble" in the second line.*

$$\text{SPEECHES} \rhd (\text{FIRST2LINES} \rhd (\text{T} \lozenge \text{B}))$$

where

$$\begin{aligned}
\text{SPEECHES} &\equiv \mathcal{I}(\text{“<speech>”}) \lozenge \mathcal{I}(\text{“</speech>”}) \\
\text{FIRST2LINES} &\equiv \mathcal{I}(\text{“<speech>”}) \lozenge \text{LINES} \lozenge \text{LINES} \\
\text{T} &\equiv \text{LINES} \rhd (\mathcal{I}(\text{“toil”}) \bigtriangledown \mathcal{I}(\text{“trouble”})) \\
\text{B} &\equiv \text{LINES} \not\rhd (\mathcal{I}(\text{“burn”}) \bigtriangledown \mathcal{I}(\text{“bubble”})) \\
\text{LINES} &\equiv \mathcal{I}(\text{“<line>”}) \lozenge \mathcal{I}(\text{“</line>”})
\end{aligned}$$

The expressions T and B select extents that match criteria on the contents of the lines. The expression FIRST2LINES evaluates to a GC-list consisting of the first two lines after the start of each speech. This expression does nothing to guarantee that both lines are contained within the speech. The outermost containment operator ensures this requirement.

5. *Find a speech by an apparition that contains "fife" and that appears in a scene along with the line "Something wicked this way comes".*

$$(\text{FIFE} \lhd (\text{S} \rhd \text{APPARITION})) \lhd (\text{SCENES} \rhd \text{B})$$

where

$$\begin{aligned}
\text{FIFE} &\equiv \text{SPEECHES} \rhd \mathcal{I}(\text{“fife”}) \\
\text{S} &\equiv \mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speech>”}) \\
\text{APPARITION} &\equiv \text{SPEAKERS} \rhd \mathcal{I}(\text{“apparition”}) \\
\text{SPEAKERS} &\equiv \mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speaker>”}) \\
\text{B} &\equiv \Sigma^5 \rhd (\text{LINES} \rhd \text{BRDBRY}) \\
\text{SPEECHES} &\equiv \mathcal{I}(\text{“<speech>”}) \lozenge \mathcal{I}(\text{“</speech>”}) \\
\text{SCENES} &\equiv \mathcal{I}(\text{“<scene>”}) \lozenge \mathcal{I}(\text{“</scene>”}) \\
\text{LINES} &\equiv \mathcal{I}(\text{“<line>”}) \lozenge \mathcal{I}(\text{“</line>”}) \\
\text{BRDBRY} &\equiv \mathcal{I}(\text{“something”}) \lozenge \mathcal{I}(\text{“wicked”}) \lozenge \mathcal{I}(\text{“this”}) \\
&\quad \lozenge \mathcal{I}(\text{“way”}) \lozenge \mathcal{I}(\text{“comes”})
\end{aligned}$$

This last example illustrates the use of Σ^n. The expression B ensures that only lines that exactly match the quote are selected. Lines such as "Something pink and wicked this way comes" are eliminated.

The query expressions given above assume a schema on the database. The expression

$$\mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speech>”})$$

occurs in several of the examples to associate speakers with their speeches. In using this expression we make the assumption that names of speakers are immediately followed by the speeches that they make. While the algebra does not depend on this assumption holding, the correctness of the query does. The schema of the database is independent of the algebra and must be described by mechanisms external to the algebra.

The algebra can be used as a tool to enforce a schema. If all speakers must be followed by a speech and all speeches must be preceded by a speaker, the following expressions must evaluate to the empty GC-list:

$$(\mathcal{I}(\text{“</speaker>”}) \lozenge \mathcal{I}(\text{“<speaker>”})) \not\rhd (\mathcal{I}(\text{“<speech>”}) \lozenge \mathcal{I}(\text{“</speech>”}))$$

$$(\mathcal{I}(\text{“</speech>”}) \lozenge \mathcal{I}(\text{“<speech>”})) \not\rhd (\mathcal{I}(\text{“<speaker>”}) \lozenge \mathcal{I}(\text{“</speaker>”}))$$

The next example illustrates a technique for handling nested structure. We assume that our collection includes Pascal program source, indexed to reflect its structure. The text model permits this indexing without any need to modify the actual source. Informally our example query is

Find triply-nested `begin`...`end` *blocks*

which may be expressed in the query algebra as

$$\text{BEGIN}2 \lozenge \text{END}2$$

where

$$
\begin{aligned}
\text{BEGIN}2 &\equiv \text{BEGIN}1 \not\triangleleft \text{BLOCK}1 \\
\text{END}2 &\equiv \text{END}1 \not\triangleleft \text{BLOCK}1 \\
\text{BLOCK}1 &\equiv \text{BEGIN}1 \lozenge \text{END}1 \\
\text{BEGIN}1 &\equiv \text{BEGIN}0 \not\triangleleft \text{BLOCK}0 \\
\text{END}1 &\equiv \text{END}0 \not\triangleleft \text{BLOCK}0 \\
\text{BLOCK}0 &\equiv \text{BEGIN}0 \lozenge \text{END}0 \\
\text{BEGIN}0 &\equiv \text{BSRC} \not\triangleleft \text{REM} \\
\text{END}0 &\equiv \text{ESRC} \not\triangleleft \text{REM} \\
\text{BSRC} &\equiv \mathcal{I}(\text{“}\texttt{begin}\text{”}) \triangleleft \text{SRC} \\
\text{ESRC} &\equiv \mathcal{I}(\text{“}\texttt{end}\text{”}) \triangleleft \text{SRC} \\
\text{REM} &\equiv \mathcal{I}(\text{“}\texttt{<comment>}\text{”}) \lozenge \mathcal{I}(\text{“}\texttt{</comment>}\text{”}) \\
\text{SRC} &\equiv \mathcal{I}(\text{“}\texttt{<program>}\text{”}) \lozenge \mathcal{I}(\text{“}\texttt{</program>}\text{”})
\end{aligned}
$$

Note that any occurrences of the words “begin” and “end” in comments are ignored.

The algebra is usable for searching data in a wide variety of formats. A great deal of on-line data is currently in the form of SQL relations. A simple representation of a SQL relation as a marked-up sequence is sufficient to permit searching of an relational database using the structure algebra of this chapter. This is particularly useful if inclusion of full-text data in relational databases becomes more prevalent (Section 2.2.4).

We treat the data in a table as if loaded sequentially in row-major order. The extent of each table is considered to be indexed with “<table>” and “</table>” markup symbols. The name of the table is indexed immediately following the table start tag and is delineated with “<tname>” and “</tname>” markup symbols. Each tuple in the relation is delineated by “<tuple>” and “</tuple>” markup symbols. Each tuple consists of a sequence of attribute name/value pairs marked-up as follows:

<name> *name* <value> *value* </value>

with the end of the name being implied by the “<value>” tag.

Assume we have a table named "`Reports`" that stores technical papers with a column for each of the title, authors, abstract and body (Figure 2.3). A query for papers written by "Clarke, Cormack and Burkowski" (a select operation) could be written as follows:

$$\text{REPORTS} \triangleright \text{CCB-AUTHORS}$$

where

$$\begin{aligned}
\text{REPORTS} &\equiv \text{TUPLES} \triangleleft \text{REPTABLE} \\
\text{TUPLES} &\equiv \mathcal{I}(\text{“<tuple>”}) \lozenge \mathcal{I}(\text{“</tuple>”}) \\
\text{REPTABLE} &\equiv \text{TABLE} \triangleright \text{R} \\
\text{TABLE} &\equiv \mathcal{I}(\text{“<table>”}) \lozenge \mathcal{I}(\text{“</table>”}) \\
\text{R} &\equiv (\mathcal{I}(\text{“<tname>”}) \lozenge \mathcal{I}(\text{“</tname>”})) \triangleright \mathcal{I}(\text{“reports”}) \\
\text{CCB-AUTHORS} &\equiv \text{AUTHORS} \triangleright (\text{VALUE} \triangleright \text{CCB}) \\
\text{AUTHORS} &\equiv (\text{NAME} \triangleright \mathcal{I}(\text{“authors”})) \lozenge \mathcal{I}(\text{“</value>”}) \\
\text{NAME} &\equiv \mathcal{I}(\text{“<name>”}) \lozenge \mathcal{I}(\text{“<value>”}) \\
\text{VALUE} &\equiv \mathcal{I}(\text{“<value>”}) \lozenge \mathcal{I}(\text{“</value>”}) \\
\text{CCB} &\equiv \mathcal{I}(\text{“clarke”}) \triangle \mathcal{I}(\text{“cormack”}) \triangle \mathcal{I}(\text{“burkowski”})
\end{aligned}$$

If we wish only the titles and abstracts of these papers (a project operation), the query would be

$$(\text{TITLE} \triangledown \text{ABSTRACT}) \triangleleft \text{CCB-REPORTS}$$

where

$$\begin{aligned}
\text{CCB-REPORTS} &\equiv \text{REPORTS} \triangleright \text{CCB-AUTHORS} \\
\text{TITLE} &\equiv (\text{NAME} \triangleright \mathcal{I}(\text{“title”})) \lozenge \mathcal{I}(\text{“</value>”}) \\
\text{ABSTRACT} &\equiv (\text{NAME} \triangleright \mathcal{I}(\text{“abstract”})) \lozenge \mathcal{I}(\text{“</value>”})
\end{aligned}$$

Finally, the operators may be used to express ordering relationships other than that of the "before" ordering operator ($\lozenge$). Given GC-lists A and B, suppose we wish to construct the GC-list C defined as follows:

$$C = \mathcal{G}\left(\{c \mid \exists\,(p,q) \in A \text{ and } \exists\,(p',q') \in B \text{ where } p < p' \text{ and } (p, \max(q,q')) \sqsubseteq c\}\right)$$

An extent in C contains an extent from each of A and B, with the starting position of the extent from A before the starting position of the extent from B. We may express the GC-list C using the query

$$C \equiv (\pi_1(A) \lozenge B) \triangle A$$

4.3 Properties

This section establishes several properties of the query algebra, including various associativity, commutativity and distributivity laws. As observed earlier, the "one of" combination operator ($\triangledown$) is similar to the boolean OR operator, the "both of" combination operator ($\triangle$) is similar to the standard boolean AND operator, while the ordering operator ($\lozenge$) generalizes concatenation. The properties of these operators are shown to be consistent with these observations. While several results from this section will be used in later proofs, the primary purpose of this section is to explore and elaborate the semantic characteristics of the algebra.

Commutativity of the combination operators follows immediately from the definitions:

Theorem 4.1 (Commutative Properties)

i) $A \triangle B = B \triangle A$

ii) $A \triangledown B = B \triangledown A$

Proof:

Immediate from the commutativity of "and" and "or" in the definitions. □

The ordering operator is, naturally enough, not commutative.

$$A \lozenge B \neq B \lozenge A$$

For example, consider $A = \{(2,2)\}$ and $B = \{(1,1),(3,3)\}$.

$$A \lozenge B = \{(2,3)\} \neq \{(1,2)\} = B \lozenge A$$

In order to establish associativity and distributivity properties of the combination and ordering operators we first establish results concerning the definitions themselves. The definitions may be

viewed as consisting of two steps: 1) a predicate that selects from the universe of all extents, followed by 2) an application of the $\mathcal{G}$ function. In order to discuss each step separately we define operators, each one corresponding to one of the ordering or combination operators, that represents only the first of these two steps. The operators are

$$\begin{aligned} A \triangle^* B &= \{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ and } \exists\, b \in B \text{ such that } b \sqsubseteq c\} \\ A \bigtriangledown^* B &= \{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ or } \exists\, b \in B \text{ such that } b \sqsubseteq c\} \\ A \Diamond^* B &= \{c \mid \exists\, (p,q) \in A \text{ and } \exists\, (p',q') \in B \text{ where } q < p' \text{ and } (p,q') \sqsubseteq c\} \end{aligned}$$

so that

$$\begin{aligned} A \triangle B &= \mathcal{G}\,(A \triangle^* B) \\ A \bigtriangledown B &= \mathcal{G}\,(A \bigtriangledown^* B) \\ A \Diamond B &= \mathcal{G}\,(A \Diamond^* B) \end{aligned}$$

The commutativity of the $\triangle^*$ and $\bigtriangledown^*$ operators is immediate from the definitions, as are the following associativity and distributivity properties:

Theorem 4.2

i) $(A \triangle^* B) \triangle^* C = A \triangle^* (B \triangle^* C)$
ii) $(A \bigtriangledown^* B) \bigtriangledown^* C = A \bigtriangledown^* (B \bigtriangledown^* C)$
iii) $(A \Diamond^* B) \Diamond^* C = A \Diamond^* (B \Diamond^* C)$

Theorem 4.3

i) $(A \bigtriangledown^* B) \triangle^* C = (A \triangle^* C) \bigtriangledown^* (B \triangle^* C)$
ii) $(A \triangle^* B) \bigtriangledown^* C = (A \bigtriangledown^* C) \triangle^* (B \bigtriangledown^* C)$
iii) $(A \bigtriangledown^* B) \Diamond^* C = (A \Diamond^* C) \bigtriangledown^* (B \Diamond^* C)$
iv) $(A \triangle^* B) \Diamond^* C = (A \Diamond^* C) \triangle^* (B \Diamond^* C)$
v) $A \Diamond^* (B \bigtriangledown^* C) = (A \Diamond^* B) \bigtriangledown^* (A \Diamond^* C)$
vi) $A \Diamond^* (B \triangle^* C) = (A \Diamond^* B) \triangle^* (A \Diamond^* C)$

A key observation concerning the three operators is that their results are insensitive to the application of the $\mathcal{G}$ function to their arguments. This property will be exploited extensively

in proofs concerning the combination and ordering operators. The following theorem formally establishes the property.

Theorem 4.4

i) $A \triangle^* B = \mathcal{G}(A) \triangle^* B$
ii) $A \triangle^* B = A \triangle^* \mathcal{G}(B)$
iii) $A \triangledown^* B = \mathcal{G}(A) \triangledown^* B$
iv) $A \triangledown^* B = A \triangledown^* \mathcal{G}(B)$
v) $A \Diamond^* B = \mathcal{G}(A) \Diamond^* B$
vi) $A \Diamond^* B = A \Diamond^* \mathcal{G}(B)$

Proof:

i) $A \triangle^* B = \mathcal{G}(A) \triangle^* B$:

If c is any element of $A \triangle^* B$ then $\exists a \in A$ and $\exists b \in B$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$. By theorem 3.3, $\exists a' \in \mathcal{G}(A)$ such that $a' \sqsubseteq a \sqsubseteq c$. Therefore, $c \in \mathcal{G}(A) \triangle^* B$, implying $A \triangle^* B \subseteq \mathcal{G}(A) \triangle^* B$.

If c is any element of $\mathcal{G}(A) \triangle^* B$ then $\exists a \in \mathcal{G}(A)$ and $\exists b \in B$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$. Since $\mathcal{G}(A) \subseteq A$, we have $a \in A$. Therefore, $c \in A \triangle^* B$, implying $\mathcal{G}(A) \triangle^* B \subseteq A \triangle^* B$.

ii) $A \triangle^* B = A \triangle^* \mathcal{G}(B)$:

Follows from part i) and the commutativity of $\triangle^*$.

iii) $A \triangledown^* B = \mathcal{G}(A) \triangledown^* B$:

If c is any element of $A \triangledown^* B$ then $\exists a \in A$ such that $a \sqsubseteq c$ or $\exists b \in B$ such that $b \sqsubseteq c$. If $\exists a \in A$ such that $a \sqsubseteq c$ then by theorem 3.3 $\exists' a \in \mathcal{G}(A)$ such that $a' \sqsubseteq a \sqsubseteq c$. Therefore, $c \in \mathcal{G}(A) \triangledown^* B$, implying $A \triangledown^* B \subseteq \mathcal{G}(A) \triangledown^* B$.

If c is any element of $\mathcal{G}(A) \triangledown^* B$ then $\exists a \in \mathcal{G}(A)$ such that $a \sqsubseteq c$ or $\exists b \in B$ such that $b \sqsubseteq c$. If $\exists a \in \mathcal{G}(A)$ such that $a \sqsubseteq c$ then since $\mathcal{G}(A) \subseteq A$ we have $a \in A$. Therefore, $c \in A \triangledown^* B$, implying $\mathcal{G}(A) \triangledown^* B \subseteq A \triangledown^* B$.

iv) $A \triangledown^* B = A \triangledown^* \mathcal{G}(B)$:

Follows from part iii) and the commutativity of $\triangledown^*$.

v) $A \Diamond^* B = \mathcal{G}(A) \Diamond^* B$:

If c is any element of $A \Diamond^* B$ then $\exists (p, q) \in A$ and $\exists (p', q') \in B$ such that $q < p'$ and $(p, q') \sqsubseteq c$. By theorem3.3, $\exists (p'', q'') \in \mathcal{G}(A)$ such that $(p'', q'') \sqsubseteq (p, q)$. Since $q'' \leq q$, we have $q'' < p' \leq q'$ and further we have $(p'', q'') \sqsubseteq (p, q') \sqsubseteq c$. Therefore, $c \in \mathcal{G}(A) \Diamond^* B$, implying $A \Diamond^* B \subseteq \mathcal{G}(A) \Diamond^* B$.

If c is any element of $\mathcal{G}(A) \Diamond^* B$ then $\exists (p, q) \in \mathcal{G}(A)$ and $\exists (p', q') \in B$ such that $q < p'$ and $(p, q') \sqsubseteq c$. Since $\mathcal{G}(A) \subseteq A$, we have $(p, q) \in A$. Therefore, $c \in A \Diamond^* B$, implying $\mathcal{G}(A) \Diamond^* B \subseteq A \Diamond^* B$.

vi) $A \Diamond^* B = A \Diamond^* \mathcal{G}(B)$:

If c is any element of $A \Diamond^* B$ then $\exists (p, q) \in A$ and $\exists (p', q') \in B$ such that $q < p'$ and $(p, q') \sqsubseteq c$. By theorem3.3, $\exists (p'', q'') \in \mathcal{G}(B)$ such that $(p'', q'') \sqsubseteq (p', q')$. Since $p' \leq p''$, we have $q < p'' \leq q''$ and further we have $(p'', q'') \sqsubseteq (p, q') \sqsubseteq c$. Therefore, $c \in A \Diamond^* \mathcal{G}(B)$, implying $A \Diamond^* B \subseteq A \Diamond^* \mathcal{G}(B)$.

If c is any element of $A \Diamond^* \mathcal{G}(B)$ then $\exists (p, q) \in A$ and $\exists (p', q') \in \mathcal{G}(B)$ such that $q < p'$ and $(p, q') \sqsubseteq c$. Since $\mathcal{G}(B) \subseteq B$, we have $(p', q') \in B$. Therefore, $c \in A \Diamond^* B$, implying $A \Diamond^* \mathcal{G}(B) \subseteq A \Diamond^* B$.

□

Proofs of associative and distributive properties are now straightforward. These proofs are very similar in character to one another. The basic proof technique is to first re-write the left-hand side of each equation as the two-step application of a predicate followed by the $\mathcal{G}$ function. Then theorem 4.4 is used to eliminate all but the outermost application of the $\mathcal{G}$ function. The underlying predicate is then manipulated directly. Finally theorem 4.4 is used to restore the application of the $\mathcal{G}$ function throughout the equation, leaving it in the desired form.

Theorem 4.5 (Associative Properties)

i) $(A \triangle B) \triangle C = A \triangle (B \triangle C)$

ii) $(A \triangledown B) \triangledown C = A \triangledown (B \triangledown C)$

iii) $(A \Diamond B) \Diamond C = A \Diamond (B \Diamond C)$

Proof:

i) $(A \triangle B) \triangle C = A \triangle (B \triangle C)$:

$$\begin{aligned}
(A \triangle B) \triangle C &= \mathcal{G}\,(\,\mathcal{G}\,(A \triangle^* B) \triangle^* C) && \textit{(Definitions)} \\
&= \mathcal{G}\,((A \triangle^* B) \triangle^* C) && \textit{(Theorem 4.4,i)} \\
&= \mathcal{G}\,(A \triangle^* (B \triangle^* C)) && \textit{(Theorem 4.2,i)} \\
&= \mathcal{G}\,(A \triangle^* \mathcal{G}\,(B \triangle^* C)) && \textit{(Theorem 4.4,ii)} \\
&= A \triangle (B \triangle C) && \textit{(Definitions)}
\end{aligned}$$

ii) $(A \triangledown B) \triangledown C = A \triangledown (B \triangledown C)$:

$$\begin{aligned}
(A \triangledown B) \triangledown C &= \mathcal{G}\,(\,\mathcal{G}\,(A \triangledown^* B) \triangledown^* C) && \textit{(Definitions)} \\
&= \mathcal{G}\,((A \triangledown^* B) \triangledown^* C) && \textit{(Theorem 4.4,iii)} \\
&= \mathcal{G}\,(A \triangledown^* (B \triangledown^* C)) && \textit{(Theorem 4.2,ii)} \\
&= \mathcal{G}\,(A \triangledown^* \mathcal{G}\,(B \triangledown^* C)) && \textit{(Theorem 4.4,iv)} \\
&= A \triangledown (B \triangledown C) && \textit{(Definitions)}
\end{aligned}$$

iii) $(A \Diamond B) \Diamond C = A \Diamond (B \Diamond C)$:

$$\begin{aligned}
(A \Diamond B) \Diamond C &= \mathcal{G}\,(\,\mathcal{G}\,(A \Diamond^* B) \Diamond^* C) && \textit{(Definitions)} \\
&= \mathcal{G}\,((A \Diamond^* B) \Diamond^* C) && \textit{(Theorem 4.4,v)} \\
&= \mathcal{G}\,(A \Diamond^* (B \Diamond^* C)) && \textit{(Theorem 4.2,iii)} \\
&= \mathcal{G}\,(A \Diamond^* \mathcal{G}\,(B \Diamond^* C)) && \textit{(Theorem 4.4,vi)} \\
&= A \Diamond (B \Diamond C) && \textit{(Definitions)}
\end{aligned}$$

□

Theorem 4.6 (Distributive Properties)

i) $(A \bigtriangledown B) \bigtriangleup C = (A \bigtriangleup C) \bigtriangledown (B \bigtriangleup C)$
ii) $(A \bigtriangleup B) \bigtriangledown C = (A \bigtriangledown C) \bigtriangleup (B \bigtriangledown C)$
iii) $(A \bigtriangledown B) \Diamond C = (A \Diamond C) \bigtriangledown (B \Diamond C)$
iv) $(A \bigtriangleup B) \Diamond C = (A \Diamond C) \bigtriangleup (B \Diamond C)$
v) $A \Diamond (B \bigtriangledown C) = (A \Diamond B) \bigtriangledown (A \Diamond C)$
vi) $A \Diamond (B \bigtriangleup C) = (A \Diamond B) \bigtriangleup (A \Diamond C)$

Proof:

i) $(A \bigtriangledown B) \bigtriangleup C = (A \bigtriangleup C) \bigtriangledown (B \bigtriangleup C)$:

$$\begin{aligned}
(A \bigtriangledown B) \bigtriangleup C &= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangledown^* B) \bigtriangleup^* C) && \textit{(Definitions)}\\
&= \mathcal{G}\,((A \bigtriangledown^* B) \bigtriangleup^* C) && \textit{(Theorem 4.4,i)}\\
&= \mathcal{G}\,((A \bigtriangleup^* C) \bigtriangledown^* (B \bigtriangleup^* C)) && \textit{(Theorem 4.3,i)}\\
&= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangleup^* C) \bigtriangledown^* \mathcal{G}\,(B \bigtriangleup^* C)) && \textit{(Theorem 4.4,iii,iv)}\\
&= (A \bigtriangleup C) \bigtriangledown (B \bigtriangleup C) && \textit{(Definitions)}
\end{aligned}$$

ii) $(A \bigtriangleup B) \bigtriangledown C = (A \bigtriangledown C) \bigtriangleup (B \bigtriangledown C)$:

$$\begin{aligned}
(A \bigtriangleup B) \bigtriangledown C &= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangleup^* B) \bigtriangledown^* C) && \textit{(Definitions)}\\
&= \mathcal{G}\,((A \bigtriangleup^* B) \bigtriangledown^* C) && \textit{(Theorem 4.4,iii)}\\
&= \mathcal{G}\,((A \bigtriangledown^* C) \bigtriangleup^* (B \bigtriangledown^* C)) && \textit{(Theorem 4.3,ii)}\\
&= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangledown^* C) \bigtriangleup^* \mathcal{G}\,(B \bigtriangledown^* C)) && \textit{(Theorem 4.4,i,ii)}\\
&= (A \bigtriangledown C) \bigtriangleup (B \bigtriangledown C) && \textit{(Definitions)}
\end{aligned}$$

iii) $(A \bigtriangledown B) \Diamond C = (A \Diamond C) \bigtriangledown (B \Diamond C)$:

$$\begin{aligned}
(A \bigtriangledown B) \Diamond C &= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangledown^* B) \Diamond^* C) && \textit{(Definitions)}\\
&= \mathcal{G}\,((A \bigtriangledown^* B) \Diamond^* C) && \textit{(Theorem 4.4,v)}\\
&= \mathcal{G}\,((A \Diamond^* C) \bigtriangledown^* (B \Diamond^* C)) && \textit{(Theorem 4.3,iii)}\\
&= \mathcal{G}\,(\,\mathcal{G}\,(A \Diamond^* C) \bigtriangledown^* \mathcal{G}\,(B \Diamond^* C)) && \textit{(Theorem 4.4,iii,iv)}\\
&= (A \Diamond C) \bigtriangledown (B \Diamond C) && \textit{(Definitions)}
\end{aligned}$$

iv) $(A \bigtriangleup B) \Diamond C = (A \Diamond C) \bigtriangleup (B \Diamond C)$:

$$\begin{aligned}
(A \bigtriangleup B) \Diamond C &= \mathcal{G}\,(\,\mathcal{G}\,(A \bigtriangleup^* B) \Diamond^* C) && \textit{(Definitions)}\\
&= \mathcal{G}\,((A \bigtriangleup^* B) \Diamond^* C) && \textit{(Theorem 4.4,v)}\\
&= \mathcal{G}\,((A \Diamond^* C) \bigtriangleup^* (B \Diamond^* C)) && \textit{(Theorem 4.3,iv)}\\
&= \mathcal{G}\,(\,\mathcal{G}\,(A \Diamond^* C) \bigtriangleup^* \mathcal{G}\,(B \Diamond^* C)) && \textit{(Theorem 4.4,i,ii)}\\
&= (A \Diamond C) \bigtriangleup (B \Diamond C) && \textit{(Definitions)}
\end{aligned}$$

v) $A \Diamond (B \bigtriangledown C) = (A \Diamond B) \bigtriangledown (A \Diamond C)$:

$$\begin{aligned} A \Diamond (B \bigtriangledown C) &= \mathcal{G}\,(A \Diamond^* \mathcal{G}\,(B \bigtriangledown^* C)) && \textit{(Definitions)} \\ &= \mathcal{G}\,(A \Diamond^* (B \bigtriangledown^* C)) && \textit{(Theorem 4.4,vi)} \\ &= \mathcal{G}\,((A \Diamond^* B) \bigtriangledown^* (A \Diamond^* C)) && \textit{(Theorem 4.3,v)} \\ &= \mathcal{G}\,(\mathcal{G}\,(A \Diamond^* B) \bigtriangledown^* \mathcal{G}\,(A \Diamond^* C)) && \textit{(Theorem 4.4,iii,iv)} \\ &= (A \Diamond B) \bigtriangledown (A \Diamond C) && \textit{(Definitions)} \end{aligned}$$

vi) $A \Diamond (B \bigtriangleup C) = (A \Diamond B) \bigtriangleup (A \Diamond C)$:

$$\begin{aligned} A \Diamond (B \bigtriangleup C) &= \mathcal{G}\,(A \Diamond^* \mathcal{G}\,(B \bigtriangleup^* C)) && \textit{(Definitions)} \\ &= \mathcal{G}\,(A \Diamond^* (B \bigtriangleup^* C)) && \textit{(Theorem 4.4,vi)} \\ &= \mathcal{G}\,((A \Diamond^* B) \bigtriangleup^* (A \Diamond^* C)) && \textit{(Theorem 4.3,vi)} \\ &= \mathcal{G}\,(\mathcal{G}\,(A \Diamond^* B) \bigtriangleup^* \mathcal{G}\,(A \Diamond^* C)) && \textit{(Theorem 4.4,i,ii)} \\ &= (A \Diamond B) \bigtriangleup (A \Diamond C) && \textit{(Definitions)} \end{aligned}$$

□

The next three results provide information about the structure of $A \bigtriangledown B$, simplifying the original definition.

Theorem 4.7

$A \bigtriangledown B \subseteq A \cup B$

Proof:

Assume $\exists c \in \mathcal{G}\,(A \bigtriangledown^* B)$ such that $c \notin A$ and $c \notin B$. Since $c \in \mathcal{G}\,(A \bigtriangledown^* B)$, we have $c \in A \bigtriangledown^* B$. Since $c \in A \bigtriangledown^* B$, $\exists a \in A$ such that $a \sqsubset c$ or $\exists b \in B$ such that $b \sqsubset c$. If $a \in A$ then $a \in A \bigtriangledown^* B$; if $b \in B$ then $b \in A \bigtriangledown^* B$. In either case c cannot be a member of $\mathcal{G}\,(A \bigtriangledown^* B)$, a contradiction. Therefore, if $c \in A \bigtriangledown B$ then $c \in A$ or $c \in B$. □

Theorem 4.8
$A \cup B \subseteq A \bigtriangledown^* B$

Proof:

Immediate from the definition of the $\bigtriangledown^*$ operator. □

Theorem 4.9
$A \bigtriangledown B = \mathcal{G}\ (A \cup B)$

Proof:

Assume $\exists c \in A \bigtriangledown B$ such that $c \notin \mathcal{G}\ (A \cup B)$. By theorem 4.7 $c \in A \cup B$. By the definition of the $\mathcal{G}$ operator $\exists c' \in A \cup B$ such that $c' \sqsubset c$. By theorem 4.8 $c' \in A \bigtriangledown^* B$. Since $c' \sqsubset c$, we have $c \notin \mathcal{G}\ (A \bigtriangledown^* B)\ (= A \bigtriangledown B)$, a contradiction.

Assume $\exists c \in \mathcal{G}\ (A \cup B)$ such that $c \notin A \bigtriangledown B\ (= \mathcal{G}\ (A \bigtriangledown^* B))$. By theorem 4.8 $c \in A \bigtriangledown^* B$, and therefore $\exists a \in A$ such that $a \sqsubset c$ or $\exists b \in B$ such that $b \sqsubset c$. In either case c cannot be an element of $\mathcal{G}\ (A \cup B)$, a contradiction. □

To complete the list of properties for the combination operators, we note that identity elements exist for both the $\triangle$ and $\bigtriangledown$ operators. If Σ^ϵ is the set of extents of smallest length and $\emptyset$ is the empty GC-list then

$$A \triangle \Sigma^\epsilon = \Sigma^\epsilon \triangle A = A$$
$$A \bigtriangledown \emptyset = \emptyset \bigtriangledown A = A$$

The containment operators exhibit fewer general properties than the combination and ordering operators. In some cases the containment operators distribute across the $\bigtriangledown$ operator:

Theorem 4.10
i) $(A \bigtriangledown B) \lhd C = (A \lhd C) \bigtriangledown (B \lhd C)$
ii) $(A \bigtriangledown B) \not\rhd C = (A \not\rhd C) \bigtriangledown (B \not\rhd C)$
iii) $A \rhd (B \bigtriangledown C) = (A \rhd B) \bigtriangledown (A \rhd C)$

Proof:

i) $(A \bigtriangledown B) \lhd C = (A \lhd C) \bigtriangledown (B \lhd C)$:

Let a be any element of $(A \bigtriangledown B) \lhd C$ such that $a \notin (A \lhd C) \bigtriangledown (B \lhd C)$. Either $a \in A$ or $a \in B$. Without loss of generality assume that $a \in A$. Since $a \in (A \bigtriangledown B) \lhd C$ and $a \in A$, we have $a \in A \lhd C$. Since $a \notin (A \lhd C) \bigtriangledown (B \lhd C)$, $\exists b \in B \lhd C$ such that $b \sqsubset a$. But then $\exists b \in B$ such that $b \sqsubset a$, implying $a \notin A \bigtriangledown B$ and therefore $a \notin (A \bigtriangledown B) \lhd C$, a contradiction.

Let a be any element of $(A \lhd C) \bigtriangledown (B \lhd C)$ such that $a \notin (A \bigtriangledown B) \lhd C$. Either $a \in A \lhd C$ or $a \in B \lhd C$. Without loss of generality assume that $a \in A \lhd C$. Since $a \notin (A \bigtriangledown B) \lhd C$ but $a \in A \lhd C$, we have $a \notin A \bigtriangledown B$. Since $a \in A$ but $a \notin A \bigtriangledown B$, $\exists b \in B$ such that $b \sqsubset a$. Since $a \in A \lhd C$, $\exists c \in C$ such that $b \sqsubset a \sqsubseteq c$. Thus $b \in B \lhd C$ and $a \notin (A \lhd C) \bigtriangledown (B \lhd C)$, a contradiction.

ii) $(A \bigtriangledown B) \not\rhd C = (A \not\rhd C) \bigtriangledown (B \not\rhd C)$:

Let a be any element of $(A \bigtriangledown B) \not\rhd C$ such that $a \notin (A \not\rhd C) \bigtriangledown (B \not\rhd C)$. Either $a \in A$ or $a \in B$. Without loss of generality assume that $a \in A$. Since $a \in (A \bigtriangledown B) \not\rhd C$ and $a \in A$, we have $a \in A \not\rhd C$. Since $a \notin (A \not\rhd C) \bigtriangledown (B \not\rhd C)$, $\exists b \in B \not\rhd C$ such that $b \sqsubset a$. But then $\exists b \in B$ such that $b \sqsubset a$, implying $a \notin A \bigtriangledown B$ and therefore $a \notin (A \bigtriangledown B) \not\rhd C$, a contradiction.

Let a be any element of $(A \not\rhd C) \bigtriangledown (B \not\rhd C)$ such that $a \notin (A \bigtriangledown B) \not\rhd C$. Either $a \in A \not\rhd C$ or $a \in B \not\rhd C$. Without loss of generality assume that $a \in A \not\rhd C$. Since $a \notin (A \bigtriangledown B) \not\rhd C$ but $a \in A \not\rhd C$, we have $a \notin A \bigtriangledown B$. Since $a \in A$ but $a \notin A \bigtriangledown B$, $\exists b \in B$ such that $b \sqsubset a$. Since $a \in (A \not\rhd C) \bigtriangledown (B \not\rhd C)$, we have $b \notin B \not\rhd C$. Therefore, $\exists c \in C$ such that $c \sqsubseteq b \sqsubset a$ and $a \notin A \not\rhd C$, a contradiction.

iii) $A \rhd (B \bigtriangledown C) = (A \rhd B) \bigtriangledown (A \rhd C)$:

Let a be any element of $A \rhd (B \bigtriangledown C)$ such that $a \notin (A \rhd B) \bigtriangledown (A \rhd C)$. Since $a \in A \rhd (B \bigtriangledown C)$, either $\exists b \in B$ such that $b \sqsubseteq a$ or $\exists c \in C$ such that $c \sqsubseteq a$. Without loss of generality assume that $\exists b \in B$ such that $b \sqsubseteq a$, implying $a \in A \rhd B$. Since $a \in A \rhd B$ but $a \notin (A \rhd B) \bigtriangledown (A \rhd C)$, $\exists a' \in A \rhd C$ such that $'a \sqsubset a$. Therefore, $\exists a' \in A$ such that $'a \sqsubset a$, and A is not a GC-list, a contradiction.

Let a be any element of $(A \rhd B) \bigtriangledown (A \rhd C)$ such that $a \notin A \rhd (B \bigtriangledown C)$. Either $a \in A \rhd B$ or $a \in A \rhd C$. Without loss of generality assume that $a \in A \rhd B$, implying $\exists b \in B$ such that $b \sqsubseteq a$. Since $a \notin A \rhd (B \bigtriangledown C)$, we have $b \notin B \bigtriangledown C$. If $b \notin B \bigtriangledown C$ then $\exists c \in B \bigtriangledown C$ such that $c \sqsubset b$. But then $c \sqsubset a$ and $a \in A \rhd (B \bigtriangledown C)$, a contradiction.

□

In the other cases, containment operators do not distribute across the $\bigtriangledown$ operator. For example, consider

$$A = \{(2,5)\}, \quad B = \{(3,4)\}, \quad C = \{(1,6)\}$$

$$\begin{aligned} B \bigtriangledown C &= \{(3,4)\} \\ A \lhd (B \bigtriangledown C) &= \emptyset \\ A \lhd B &= \emptyset \\ A \lhd C &= \{(2,5)\} \\ (A \lhd B) \bigtriangledown (A \lhd C) &= \{(2,5)\} \\ A \lhd (B \bigtriangledown C) &\neq (A \lhd B) \bigtriangledown (A \lhd C) \end{aligned}$$

Theorem 4.11
$A \rhd (B \bigtriangleup C) = (A \rhd B) \rhd C$

Proof:

If $a \in A \rhd (B \bigtriangleup C)$ then $\exists d \in B \bigtriangleup C$ such that $d \sqsubseteq a$. Since $d \in B \bigtriangleup C$, $\exists b \in B$ such that $b \sqsubseteq d \sqsubseteq a$ and $\exists c \in C$ such that $c \sqsubseteq d \sqsubseteq a$. Since $b \sqsubseteq a$, $a \in A \rhd B$ and, since $c \sqsubseteq a$, $a \in (A \rhd B) \rhd C$.

If $a \in (A \rhd B) \rhd C$ then $\exists b \in B$ such that $b \sqsubseteq a$ and $\exists c \in C$ such that $c \sqsubseteq a$, implying $a \in B \bigtriangleup^* C$. Therefore $\exists d \in B \bigtriangleup C$ such that $d \sqsubseteq a$, implying $a \in A \rhd (B \bigtriangleup C)$.

□

Finally, the containment operators exhibit a limited version of commutativity, commutativity of containment criteria applied to a particular GC-list.

Theorem 4.12
$(A \circ B) \bullet C = (A \bullet C) \circ B$
where $\circ$ *and* $\bullet$ *are any of the containment operators:* $\lhd$, $\rhd$, $\not\lhd$, *and* $\not\rhd$.

Proof:

Immediate from the definitions of the containment operators. □

4.4 A Framework for Implementation

Start points and end points place identical total orders on the elements of a GC-list (theorem 3.1). We exploit this total order to develop a framework for efficiently implementing the algebra. The approach consists of indexing into GC-lists; the total order is used as the basis for this indexing. Given a GC-list and a position in the database we index into the GC-list to find the extent that is in some sense "closest to" that position in the database. We begin with an example and follow this with a formal exposition of the framework.

Consider evaluating the expression $A \Diamond B$ (see Figure 4.2). An extent from the resultant GC-list starts with an element from A and ends with an element from B. Suppose (p, q) is the first extent in A. If (p', q') is the first extent from B with $p' > q$ then q' must be the end of the first extent of $A \Diamond B$. We index into B to find the first extent with $p' > q$. The last extent from A that ends before p' starts the first extent of $A \Diamond B$. We index into A to find the greatest extent (p'', q'') where $q'' < p'$. The extent (p'', q') is the first solution to $A \Diamond B$. Indexing first into B and then into A in this manner gives us the first extent in $A \Diamond B$ directly in two steps. The next solution to $A \Diamond B$ begins after p''. We index into A to produce the first extent after p''. This procedure of successively indexing into A and B can be continued to find the remaining extents in $A \Diamond B$.

The implementation framework consists of four access functions that allow indexing into GC-lists in various ways. Each of the access functions represents a variation on the notion of "closest extent" in a GC-list to a specified position in the database. We implement the four access functions for each operator in our algebra using the access functions of its operands.

The access function $\tau(S, k)$ represents the first extent in the GC-list S starting at or after the position k:

$$\tau(S,k) = \begin{cases} (p,q) & \text{if } \exists\ (p,q) \in S \text{ such that } k \le p \\ & \quad \text{and } \nexists\ (p',q') \in S \text{ such that } k \le p' < p \\ (\infty,\infty) & \text{if } \nexists\ (p,q) \in S \text{ such that } k \le p \end{cases}$$

The access function $\rho(S, k)$ represents the first extent in S ending at or after the position k:

$$\rho(S,k) = \begin{cases} (p,q) & \text{if } \exists\ (p,q) \in S \text{ such that } k \le q \\ & \quad \text{and } \nexists\ (p',q') \in S \text{ such that } k \le q' < q \\ (\infty,\infty) & \text{if } \nexists\ (p,q) \in S \text{ such that } k \le q \end{cases}$$

The access functions $\tau'(S, k)$ and $\rho'(S, k)$ are the converses of τ and ρ. The access func-

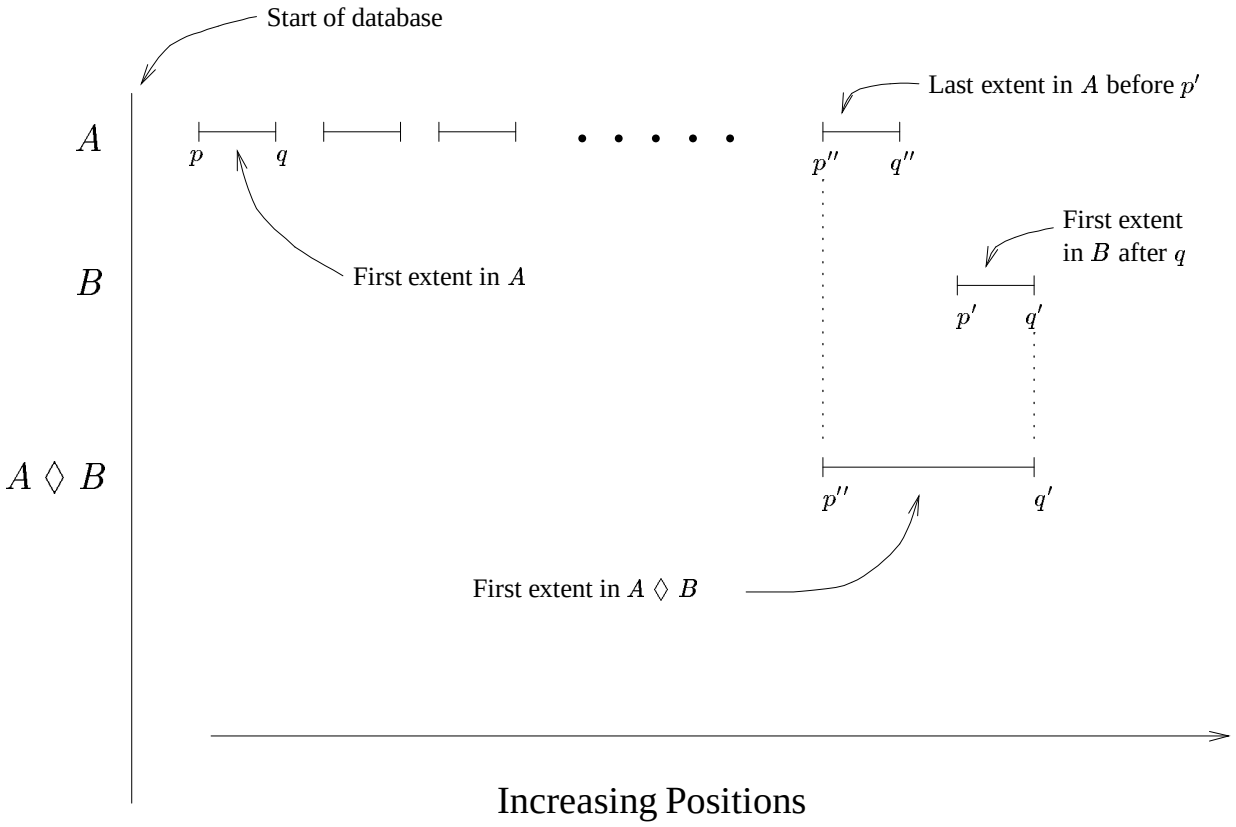


Figure 4.2: Evaluating $A \Diamond B$.

tion $\tau'(S,k)$ represents the last extent in S ending at or before the position k; the access function $\rho'(S,k)$ represents the last extent in S starting at or before the position k:

$$\tau'(S,k) = \begin{cases} (p,q) & \text{if } \exists\ (p,q) \in S \text{ such that } k \geq q \\ & \quad \text{and } \nexists\ (p',q') \in S \text{ such that } k \geq q' > q \\ (-\infty,-\infty) & \text{if } \nexists\ (p,q) \in S \text{ such that } k \geq q \end{cases}$$

$$\rho'(S,k) = \begin{cases} (p,q) & \text{if } \exists\ (p,q) \in S \text{ such that } k \geq p \\ & \quad \text{and } \nexists\ (p',q') \in S \text{ such that } k \geq p' > p \\ (-\infty,-\infty) & \text{if } \nexists\ (p,q) \in S \text{ such that } k \geq p \end{cases}$$

In these definitions, ∞ and $-\infty$ are formal symbols defined such that $k < \infty$ and $k > -\infty$, for any position k in the database. Figure 4.3 provides an illustrative example of the differences between solutions to the various access functions about a specific database position.

Figure 4.4, Figure 4.5 and Figure 4.6 give definitions of τ and ρ over the operators when $k < \infty$. For simplicity, the case of $k = \infty$ is omitted from the figures. In that case we have:

$$\tau(S,\infty) = \rho(S,\infty) = (\infty,\infty).$$

Figure 4.7, Figure 4.8 and Figure 4.9 give definitions of τ' and ρ' over the operators when $k > -\infty$. For simplicity, the case of $k = -\infty$ is omitted from the figures. In that case we have:

$$\tau'(S,-\infty) = \rho'(S,-\infty) = (-\infty,-\infty).$$

The equations also require that

$$\infty + \epsilon = \infty$$

$$\infty - \epsilon = \infty$$

$$-\infty + \epsilon = -\infty$$

$$-\infty - \epsilon = -\infty$$

The notation used in the figures is loosely based on the functional programming language ML [75]. An expression of the form

let *definitions* **in** *expression*

yields the value of the expression following the **in**, evaluated in the context of the definitions

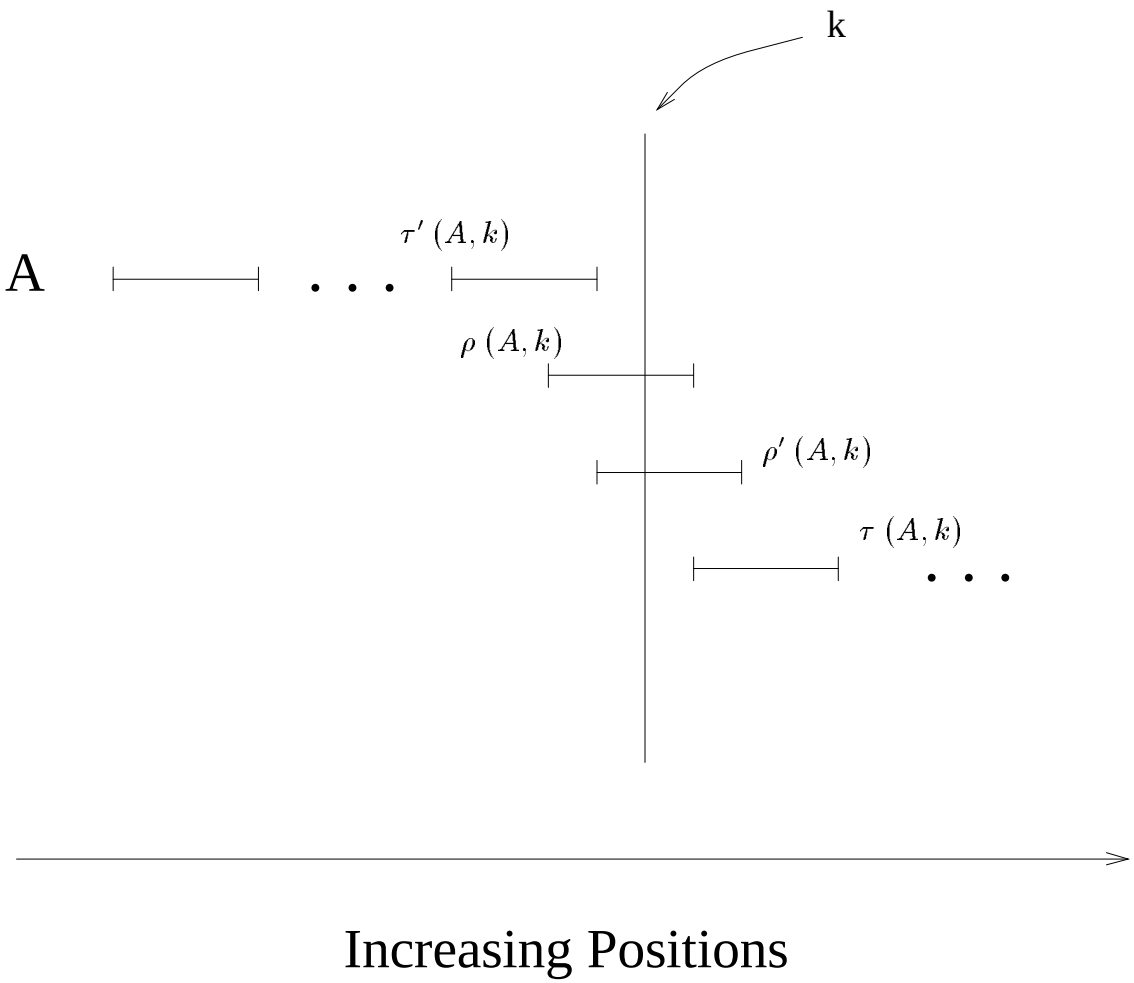


Figure 4.3: Comparison of the access functions.

following the **let**. A conditional expression

if *condition* **then** *expression* **else** *expression*

evaluates to the expression following the **then** if the Boolean condition following the **if** is true, and evaluates to the expression following the **else** if the Boolean condition is false.

We examine in detail the equation for $\tau\,(A \lhd B, k)$ (see Figure 4.10). This equation yields the first element of $A \lhd B$ that starts at or after the position k. The extent $(p, q) = \tau\,(A, k)$ is taken as a candidate solution. For (p, q) to be the solution it must be contained in an extent of B. An extent of B containing (p, q) must end at or after q. The first such extent is $(p', q') = \rho\,(B, q)$. There are now two cases: 1) If $p' \leq p$ then (p, q) is contained in (p', q') and (p, q) is the solution to $\tau\,(A \lhd B, k)$. 2) Otherwise $p' > p$ and (p, q) is not contained in (p', q'). In this case, (p, q) is not contained in any extent of B. For if there existed an extent (p'', q'') in B that contained (p, q) we would have:

$p' > p$	since (p', q') does not contain (p, q)
$p \geq p''$	since (p'', q'') contains (p, q)
$p'' \geq p'$	since (p'', q'') is after (p', q') in the GC-list B

This is a contradiction and (p, q) is not a solution to $\tau\,(A \lhd B, k)$. The solution to $\tau\,(A \lhd B, k)$ must start at or after p'. Thus, $\tau\,(A \lhd B, k) = \tau\,(A \lhd B, p')$.

Following this approach we may formally demonstrate the correctness of the equations. The associated theorems and proofs are the subject of Section 4.5.

Definitions of the access functions for the projection operators and for fixed intervals may also be given. For the projection operators the equations are:

$$\begin{aligned} \tau\,(\pi_1(A), k) &= \rho\,(\pi_1(A), k) \\ &= \pi_1(\,\tau\,(A, k)) \end{aligned}$$

$$\begin{aligned} \tau\,(\pi_2(A), k) &= \rho\,(\pi_2(A), k) \\ &= \pi_2(\,\rho\,(A, k)) \end{aligned}$$

$$\begin{aligned} \tau'\,(\pi_1(A), k) &= \rho'\,(\pi_1(A), k) \\ &= \pi_1(\,\rho'\,(A, k)) \end{aligned}$$

Containment

$\tau\,(A \lhd B, k) =$
let
$(p, q) = \tau\,(A, k)$
$(p', q') = \rho\,(B, q)$
in
if $p' \leq p$ **then**
(p, q)
else
$\tau\,(A \lhd B, p')$

$\rho\,(A \lhd B, k) =$
let
$(p, q) = \rho\,(A, k)$
in
$\tau\,(A \lhd B, p)$

$\tau\,(A \rhd B, k) =$
let
$(p, q) = \tau\,(A, k)$
in
$\rho\,(A \rhd B, q)$

$\rho\,(A \rhd B, k) =$
let
$(p, q) = \rho\,(A, k)$
$(p', q') = \tau\,(B, p)$
in
if $q' \leq q$ **then**
(p, q)
else
$\rho\,(A \rhd B, q')$

$\tau\,(A \not\lhd B, k) =$
let
$(p, q) = \tau\,(A, k)$
$(p', q') = \rho\,(B, q)$
in
if $p' > p$ **then**
(p, q)
else
$\rho\,(A \not\lhd B, q' + \epsilon)$

$\rho\,(A \not\lhd B, k) =$
let
$(p, q) = \rho\,(A, k)$
in
$\tau\,(A \not\lhd B, p)$

$\tau\,(A \not\rhd B, k) =$
let
$(p, q) = \tau\,(A, k)$
in
$\rho\,(A \not\rhd B, q)$

$\rho\,(A \not\rhd B, k) =$
let
$(p, q) = \rho\,(A, k)$
$(p', q') = \tau\,(B, p)$
in
if $q' > q$ **then**
(p, q)
else
$\tau\,(A \not\rhd B, p' + \epsilon)$

Figure 4.4: τ and ρ for the containment operators.

Combination

$\tau\,(A \triangle B, k) =$
let
$(p, q) = \tau\,(A, k)$
$(p', q') = \tau\,(B, k)$
$(p'', q'') = \tau'\,(A, \max(q, q'))$
$(p''', q''') = \tau'\,(B, \max(q, q'))$
in
$(\min(p'', p'''),\ \max(q'', q'''))$

$\rho\,(A \triangle B, k) =$
let
$(p, q) = \tau'\,(A \triangle B, k - \epsilon)$
in
$\tau\,(A \triangle B, p + \epsilon)$

$\tau\,(A \triangledown B, k) =$
let
$(p, q) = \tau\,(A, k)$
$(p', q') = \tau\,(B, k)$
in
if $q < q'$ **then**
(p, q)
else if $q > q'$ **then**
(p', q')
else
$(\max(p, p'),\ q)$

$\rho\,(A \triangledown B, k) =$
let
$(p, q) = \tau'\,(A \triangledown B, k - \epsilon)$
in
$\tau\,(A \triangledown B, p + \epsilon)$

Figure 4.5: τ and ρ for the combination operators.

Ordering

$\tau\ (A \Diamond B, k) =$
let
$(p, q) = \tau\ (A, k)$
$(p', q') = \tau\ (B, q + \epsilon)$
$(p'', q'') = \tau'\ (A, p' - \epsilon)$
in
(p'', q')

$\rho\ (A \Diamond B, k) =$
let
$(p, q) = \tau'\ (A \Diamond B, k - \epsilon)$
in
$\tau\ (A \Diamond B, p + \epsilon)$

Figure 4.6: τ and ρ for the ordering operator.

Containment

$\tau'(A \lhd B, k) =$
let
$(p,q) = \tau'(A,k)$
$(p',q') = \rho'(B,p)$
in
if $q' \geq q$ **then**
(p,q)
else
$\tau'(A \lhd B, q')$

$\rho'(A \lhd B, k) =$
let
$(p,q) = \rho'(A,k)$
in
$\tau'(A \lhd B, q)$

$\rho'(A \rhd B, k) =$
let
$(p,q) = \rho'(A,k)$
$(p',q') = \tau'(B,q)$
in
if $p' \geq p$ **then**
(p,q)
else
$\rho'(A \rhd B, p')$

$\tau'(A \rhd B, k) =$
let
$(p,q) = \tau'(A,k)$
in
$\rho'(A \rhd B, p)$

$\tau'(A \not\lhd B, k) =$
let
$(p,q) = \tau'(A,k)$
$(p',q') = \rho'(B,p)$
in
if $q' < q$ **then**
(p,q)
else
$\rho'(A \not\lhd B, p' - \epsilon)$

$\rho'(A \not\lhd B, k) =$
let
$(p,q) = \rho'(A,k)$
in
$\tau'(A \not\lhd B, q)$

$\rho'(A \not\rhd B, k) =$
let
$(p,q) = \rho'(A,k)$
$(p',q') = \tau'(B,q)$
in
if $p' < p$ **then**
(p,q)
else
$\tau'(A \not\rhd B, q' - \epsilon)$

$\tau'(A \not\rhd B, k) =$
let
$(p,q) = \tau'(A,k)$
in
$\rho'(A \not\rhd B, p)$

Figure 4.7: τ' and ρ' for the containment operators.

Combination

$\tau'(A \triangle B, k) =$
let
$(p, q) = \tau'(A, k)$
$(p', q') = \tau'(B, k)$
$(p'', q'') = \tau(A, \min(p, p'))$
$(p''', q''') = \tau(B, \min(p, p'))$
in
$(\min(p'', p'''),\ \max(q'', q'''))$

$\rho'(A \triangle B, k) =$
let
$(p, q) = \tau(A \triangle B, k + \epsilon)$
in
$\tau'(A \triangle B, q - \epsilon)$

$\tau'(A \triangledown B, k) =$
let
$(p, q) = \tau'(A, k)$
$(p', q') = \tau'(B, k)$
in
if $p > p'$ **then**
(p, q)
else if $p < p'$ **then**
(p', q')
else
$(p,\ \min(q, q'))$

$\rho'(A \triangledown B, k) =$
let
$(p, q) = \tau(A \triangledown B, k + \epsilon)$
in
$\tau'(A \triangledown B, q - \epsilon)$

Figure 4.8: τ' and ρ' for the combination operators.

Ordering

$$
\begin{array}{l}
\tau'\left(A \lozenge B, k\right) = \\
\quad \textbf{let} \\
\quad\quad (p, q) = \tau'\left(A, k\right) \\
\quad\quad (p', q') = \tau'\left(B, p - \epsilon\right) \\
\quad\quad (p'', q'') = \tau\left(A, q' + \epsilon\right) \\
\quad \textbf{in} \\
\quad\quad (p', q'')
\end{array}
$$

$$
\begin{array}{l}
\rho'\left(A \lozenge B, k\right) = \\
\quad \textbf{let} \\
\quad\quad (p, q) = \tau\left(A \lozenge B, k + \epsilon\right) \\
\quad \textbf{in} \\
\quad\quad \tau'\left(A \lozenge B, q - \epsilon\right)
\end{array}
$$

Figure 4.9: τ' and ρ' for the ordering operator.

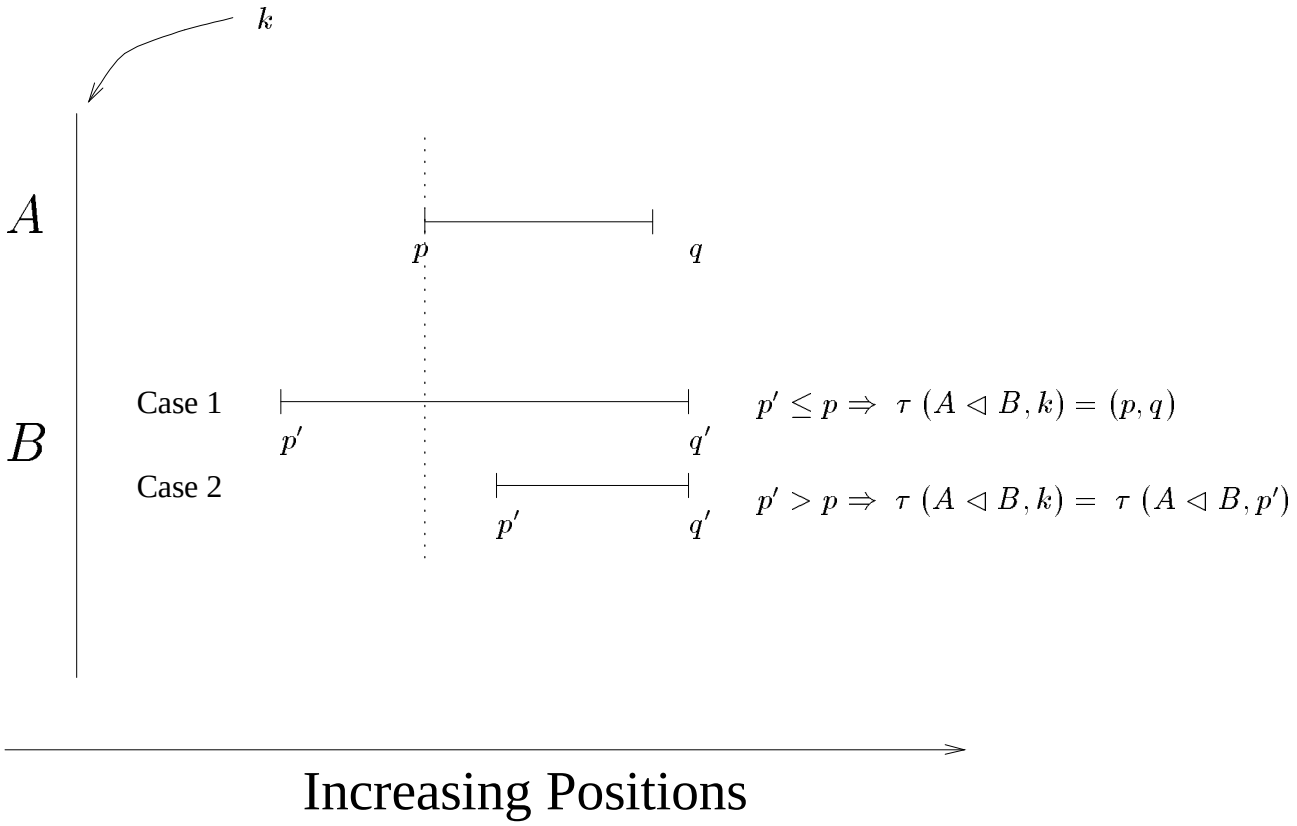


Figure 4.10: Evaluating $A \triangleleft B$.

$$\begin{aligned}\tau'(\pi_2(A),k) &= \rho'(\pi_2(A),k)\\ &= \pi_2(\tau'(A,k))\end{aligned}$$

For fixed size extents the equations are:

$$\begin{aligned}\tau(\Sigma^x,k) &= (k,k+x-\epsilon)\\ \rho(\Sigma^x,k) &= (k-x+\epsilon,k)\\ \tau'(\Sigma^x,k) &= \rho(\Sigma^x,k)\\ \rho'(\Sigma^x,k) &= \tau(\Sigma^x,k)\end{aligned}$$

Finally, we note that the access functions themselves may be expressed in terms of the query algebra. Let *first* and *last* be functions over GC-lists defined as follows:

$$\begin{aligned}\mathit{first}(S) &\equiv S \lhd (\{(-\infty,-\infty)\} \lozenge S)\\ \mathit{last}(S) &\equiv S \lhd (S \lozenge \{(\infty,\infty)\})\end{aligned}$$

The expression $\mathit{first}(S)$ represents the GC-list containing only the first element of S, and the expression $\mathit{last}(S)$ represents the GC-list containing only the last element of S. The access functions may then be expressed as follows:

$$\begin{aligned}\tau(A,k) &\equiv \mathit{first}(A' \rhd \pi_2(\{(k-\epsilon,k-\epsilon)\} \lozenge \pi_1(A')))\\ \rho(A,k) &\equiv \mathit{first}(A' \rhd \pi_2(\{(k-\epsilon,k-\epsilon)\} \lozenge \pi_2(A')))\\ \tau'(A,k) &\equiv \mathit{last}(A' \rhd \pi_1(\pi_2(A') \lozenge \{(k+\epsilon,k+\epsilon)\}))\\ \rho'(A,k) &\equiv \mathit{last}(A' \rhd \pi_1(\pi_1(A') \lozenge \{(k+\epsilon,k+\epsilon)\}))\end{aligned}$$

where

$$A' \equiv \{(-\infty,-\infty)\} \bigtriangledown A \bigtriangledown \{(\infty,\infty)\}$$

4.4.1 Index Organization

Standard data structures for inverted lists may be used to build implementations of τ and ρ for the database index function $\mathcal{I}$ [66, pages 552–554] [38]. Figure 4.11 shows the abstract organization of an inverted list data structure. The dictionary maps each index symbol into a range in the index. For each index symbol, the index contains a sorted *postings list* of database positions where the

symbol occurs. For a particular symbol, a binary search implements the four access functions with $O(\log m)$ efficiency, where m is the length of the postings list for the symbol, assuming that an initial dictionary lookup of symbols is performed before query evaluation is begun. Using a binary search, this initial dictionary lookup for each symbol requires $O(\log |\Sigma|)$ time, where $|\Sigma|$ is the number of symbols in the index alphabet. Other data structuring techniques — B-trees [66, pages 473–479], surrogate subsets [20], and dynamic update [26] — may be used to provide $O(\log m)$ implementations that additionally permit efficient insertions and deletions.

In practice, the index and dictionary will be at least partially stored on secondary storage, usually a magnetic disk or a CD-ROM. It is then important to consider the number of disk read operations separately from the number of in-memory operations. Appropriate data organization techniques allow the access functions to be implemented with a single disk operation, even when the dictionary is considerably larger than available memory resources [26]. The organization permits direct indexing to the position in a postings list that contains the solution to an access function. If the postings list for a particular symbol is of moderate size, small enough to be brought into memory in its entirety, the same organization techniques allow the postings list for a particular symbol to be brought into memory with a single disk operation. The organization further permits dynamic insertions and deletions of index information.

Under the organization, disk storage is divided into index blocks of equal size. Together, the index blocks make up the *disk index*, which combines the functions of the dictionary and index of figure 4.11. The disk index contains both index symbols and postings. Each index block contains one or more *index entries*. Each index entry consists of a term symbol followed by an occurrence count and an occurrence list, indicating positions in the database where the term occurs (Figure 4.12). By using the occurrence counts as relative pointers, we may treat an index block as a linked list of entries.

Overall, the index is ordered first by term symbol and then by database position, and divided into index blocks as appropriate. If an entry for a particular term symbol does not fit in a single index block, it is divided into multiple entries and stored in a range of index blocks. For update purposes, disk space allocated for index blocks is treated as a circular array. The disk index may start at any point in the circular array and does not usually fill it. The dynamic update algorithm operates by modifying the disk index according to an update list and writing the modified disk index into the unused portion of the array, releasing storage used by the unmodified index as it is modified and written back (Figure 4.13).

An *index map* is stored in main memory, with each entry in the index map describing a block of the disk index. Each entry contains two fields describing a block of the disk index. The first

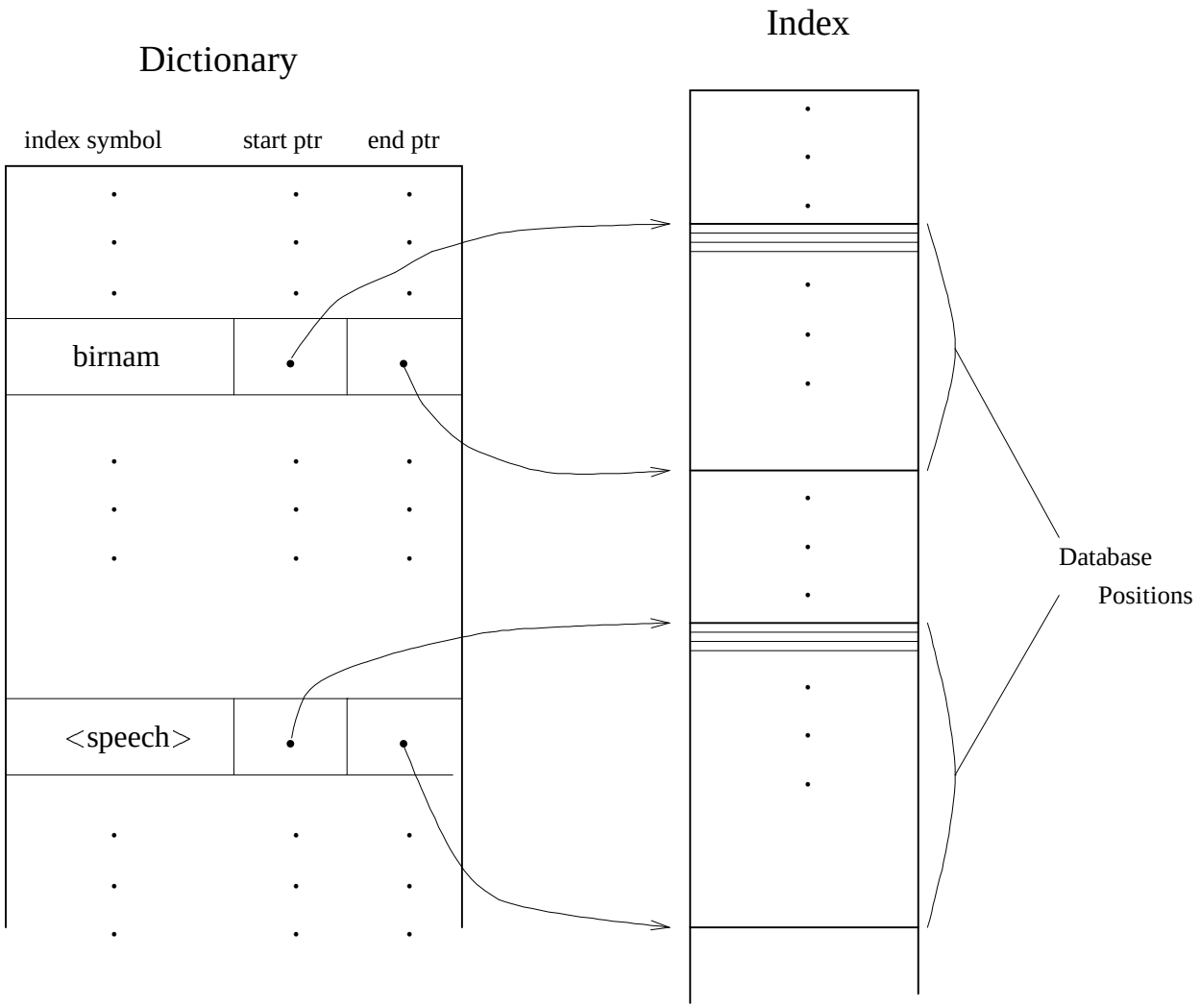


Figure 4.11: Inverted lists.

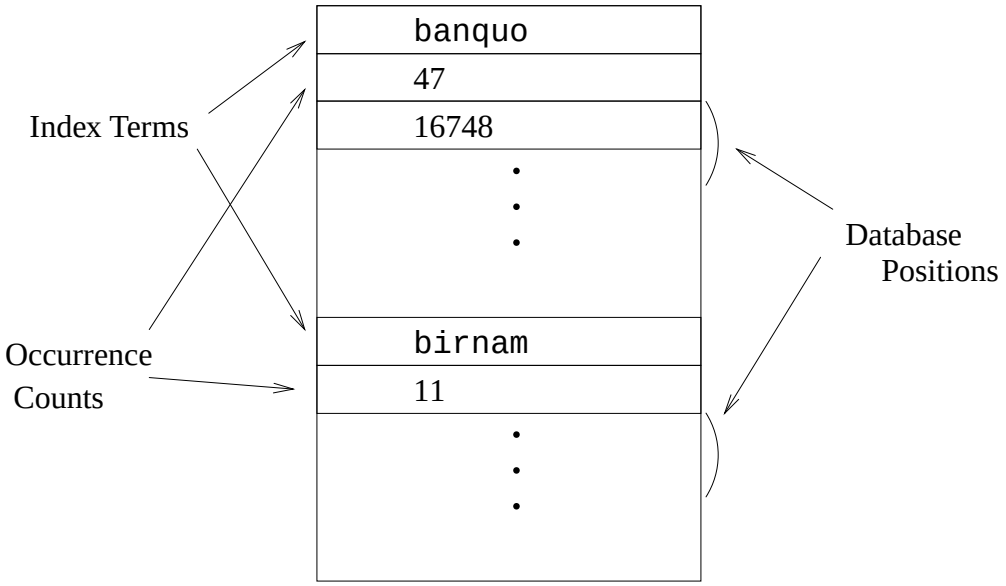


Figure 4.12: Index block organization.

field of an index map entry contains the first term symbol indexed in the block, and the second field contains the first posting for this term symbol indexed by the block. A binary search of the index map determines the range in the disk index containing the postings list for any particular symbol. If it is of small or moderate size, this postings list may then be read with a single disk access. If the postings list is long, and cannot be read in its entirety, then a further binary search of the index map may be used to choose the particular block that contains the solution to an access function for the symbol.

By slightly extending the data structures discussed in this section, it is possible to store any GC-list as an inverted list. Each index element would contain the start and end position for an extent. Since the start and end positions place identical total orders on the elements of a GC-list, any data structure usable to implement access functions for index symbols will serve equally well to implement access functions for GC-lists. In this way, GC-lists for frequently-posed queries may be pre-computed and stored as part of the database.

It is particularly useful to store GC-lists for structural elements — speeches, for example —in this form.

$$\text{SPEECHES} \equiv \mathcal{I}(\text{“<speech>”}) \lozenge \mathcal{I}(\text{“</speech>”})$$

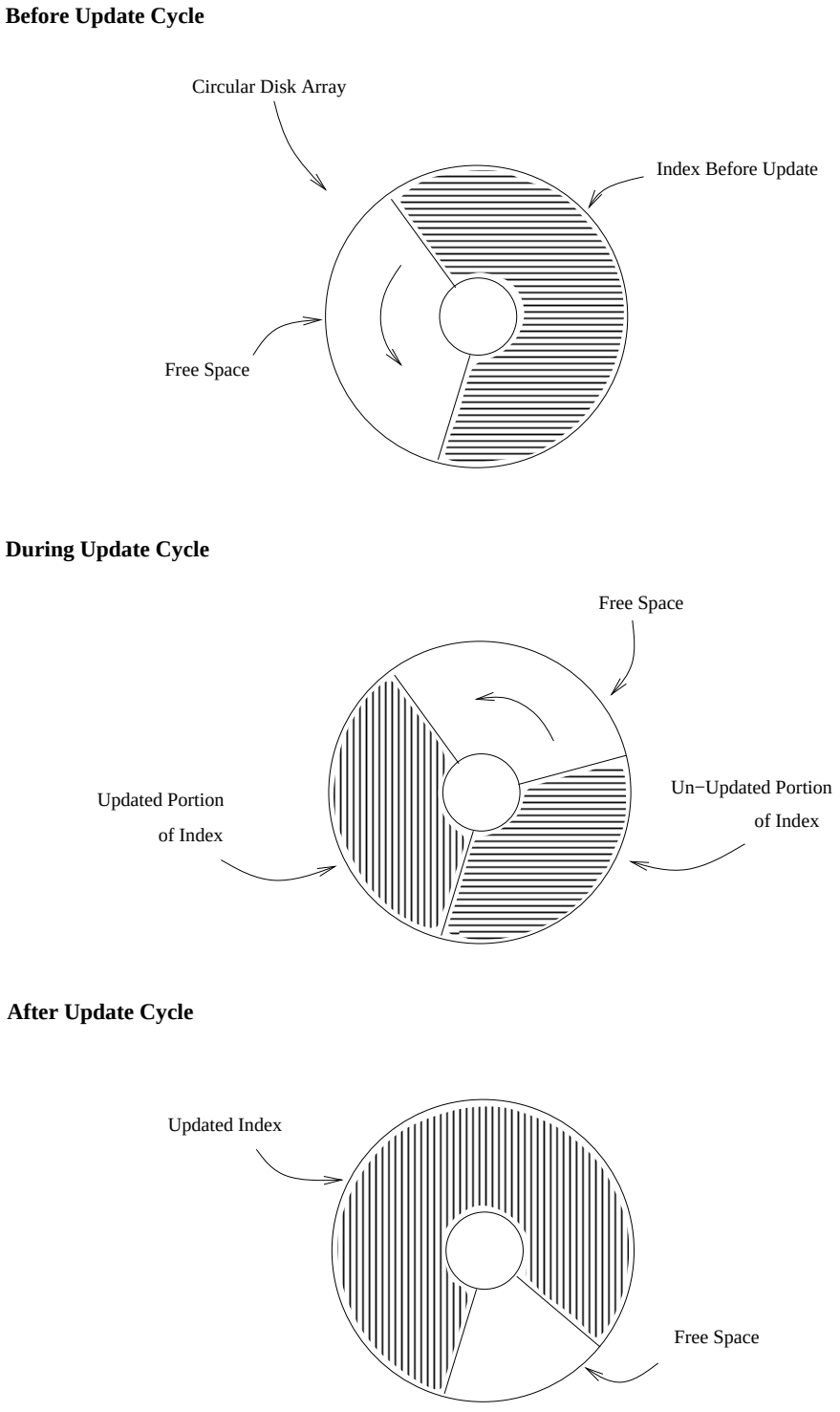


Figure 4.13: Overview of an update cycle.

Its reasonable to expect that such queries will be frequently used, and in these cases the work involved in solving for the structural element is avoided. In addition, the actual markup symbols need not be indexed since the projection operators may be used to extract this information. In the case of speeches,

$$\mathcal{I}\,(\text{“<speech>”}) \equiv \pi_1(\text{SPEECHES})$$

and

$$\mathcal{I}\,(\text{“</speech>”}) \equiv \pi_2(\text{SPEECHES})$$

4.5 Correctness of the Framework

Theorem 4.13
The equations of Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 correctly calculate τ, τ', ρ and ρ' for the operators of Figure 4.1.

Proof:
The proof addresses in detail only the equations of Figure 4.4, Figure 4.5 and Figure 4.6, which calculate τ and ρ. Since τ' and ρ' correspond to τ and ρ under a reversed ordering, the proof of the remaining equations may be derived in a straightforward fashion.

We prove the equations for GC-lists extended to include the extents $(-\infty, -\infty)$ and (∞, ∞) as members. Use of these extended GC-lists make the special handling of the infinity cases in the definitions of τ, ρ, τ' and ρ' redundant.

The equations are addressed in order, giving fourteen cases in total.

1) $\tau\,(A \lhd B, k)$ (Figure 4.4):

(This case has been informally argued in the previous section.)

Since $A \lhd B \subseteq A$, $\tau\,(A \lhd B, k) = \tau\,(A, k)$ if and only if $\exists\ (p', q') \in B$ such that $\tau\,(A, k) \sqsubseteq (p', q')$. Following the equation, let $(p, q) = \tau\,(A, k)$ and let $(p', q') = \rho\,(B, q)$. By the definition of ρ, we have $q' \geq q$. Thus, if $p' \leq p$ then $(p, q) \sqsubseteq (p', q')$ and $\tau\,(A \lhd B, k) = \tau\,(A, k)$.

Now, suppose $p' > p$. Assume $\exists\ (p'', q'') \in B$ such that $(p, q) \sqsubseteq (p'', q'')$, implying that $p'' < p'$. By the definition of ρ, if $q'' \geq q$ then $q'' > q'$. Therefore $(p', q') \sqsubset (p'', q'')$, contradicting for B the defining property of GC-lists. Finally, $\forall\ (p''', q''') \in A$ where $p < p''' < p'$, we have $(p''', q''') \neq \tau\,(A \lhd B, k)$ since $\rho\,(B, q''') \geq (p', q')$. Therefore $\tau\,(A \lhd B, k) = \tau\,(A \lhd B, p')$.

2) $\rho\,(A \lhd B, k)$ (Figure 4.4):

Let $(p,q) = \rho\,(A,k)$ and let $(p',q') = \rho\,(A \lhd B,k)$. Since $A \lhd B \subseteq A$, we have $(p',q') \geq (p,q)$. By the definition of ρ, $\nexists\,(p'',q'') \in A \lhd B$ such that $k \leq q < q'' < q'$. Since A is a GC-list it follows that $\nexists\,(p'',q'') \in A \lhd B$ such that $p < p'' < p'$. Thus from the definition of τ, we have $(p',q') = \tau\,(A \lhd B, p)$.

3) $\tau\,(A \rhd B, k)$ (Figure 4.4):

The argument is similar to the previous case. Let $(p,q) = \tau\,(A,k)$ and let $(p',q') = \tau\,(A \rhd B,k)$. Since $A \rhd B \subseteq A$, we have $(p',q') \geq (p,q)$. By the definition of τ, $\nexists\,(p'',q'') \in A \rhd B$ such that $k \leq p < p'' < p'$. Since A is a GC-list it follows that $\nexists\,(p'',q'') \in A \rhd B$ such that $q < q'' < q'$. Thus from the definition of ρ, we have $(p',q') = \rho\,(A \rhd B, q)$.

4) $\rho\,(A \rhd B, k)$ (Figure 4.4):

The argument is similar to that for $\tau\,(A \lhd B, k)$.

Since $A \rhd B \subseteq A$, $\rho\,(A \rhd B,k) = \rho\,(A,k)$ if and only if $\exists\,(p',q') \in B$ such that $(p',q') \sqsubseteq \rho\,(A,k)$. Following the equation, let $(p,q) = \rho\,(A,k)$ and let $(p',q') = \tau\,(B,p)$. By the definition of τ, we have $p' \geq p$. Thus, if $q' \leq q$ then $(p',q') \sqsubseteq (p,q)$ and $\rho\,(A \rhd B,k) = \rho\,(A,k)$.

Now, suppose $q' > q$. Assume $\exists\,(p'',q'') \in B$ such that $(p'',q'') \sqsubseteq (p,q)$, implying that $q'' < q'$. By the definition of τ, if $p'' \geq p$ then $p'' > p'$. Therefore $(p'',q'') \sqsubset (p',q')$, contradicting for B the defining property of GC-lists. Finally, $\forall\,(p''',q''') \in A$ where $q < q''' < q'$, we have $(p''',q''') \neq \rho\,(A \rhd B,k)$ since $\tau\,(B,p''') \geq (p',q')$. Therefore $\rho\,(A \rhd B,k) = \rho\,(A \rhd B,q')$.

5) $\tau\,(A \ntriangleleft B, k)$ (Figure 4.4):

Since $A \ntriangleleft B \subseteq A$, $\tau\,(A \ntriangleleft B,k) = \tau\,(A,k)$ if and only if $\nexists\,(p',q') \in B$ such that $\tau\,(A,k) \sqsubseteq (p',q')$. Following the equation, let $(p,q) = \tau\,(A,k)$ and let $(p',q') = \rho\,(B,q)$. By the definition of ρ, we have $q' \geq q$. Thus, if $p' \leq p$ then $(p,q) \sqsubseteq (p',q')$ and $\tau\,(A \ntriangleleft B,k) \neq \tau\,(A,k)$. Indeed $\forall\,(p'',q'') \in A$ where $q < q'' \leq q'$ and $p' \leq p < p''$, we have $(p'',q'') \neq \tau\,(A \ntriangleleft B,k)$. Therefore, and from the definition of ϵ, $\tau\,(A \ntriangleleft B,k) = \rho\,(A \ntriangleleft B, q'+\epsilon)$.

Now, suppose $p' > p$. Assume $\exists\,(p''',q''') \in B$ such that $(p,q) \sqsubseteq (p''',q''')$, implying that $p''' < p'$. By the definition of ρ, if $q''' \geq q$ then $q''' > q'$. Therefore $(p',q') \sqsubset (p''',q''')$, contradicting for B the defining property of GC-lists.

6) $\rho\,(A \ntriangleleft B, k)$ (Figure 4.4):

The argument is similar to that for $\rho\,(A \lhd B,k)$ and $\tau\,(A \rhd B,k)$. Let $(p,q) = \rho\,(A,k)$ and let $(p',q') = \rho\,(A \ntriangleleft B,k)$. Since $A \ntriangleleft B \subseteq A$, we have $(p',q') \geq (p,q)$. By the definition

of ρ, $\nexists\ (p'', q'') \in A \not\lhd B$ such that $k \leq q < q'' < q'$. Since A is a GC-list it follows that $\nexists\ (p'', q'') \in A \lhd B$ such that $p < p'' < p'$. Thus from the definition of τ, we have $(p', q') = \tau\ (A \not\lhd B, p)$.

7) $\tau\ (A \not\rhd B, k)$ (Figure 4.4):

The argument is similar to the previous case. Let $(p, q) = \tau\ (A, k)$ and let $(p', q') = \tau\ (A \not\rhd B, k)$. Since $A \not\rhd B \subseteq A$, we have $(p', q') \geq (p, q)$. By the definition of τ, $\nexists\ (p'', q'') \in A \not\rhd B$ such that $k \leq p < p'' < p'$. Since A is a GC-list it follows that $\nexists\ (p'', q'') \in A \not\rhd B$ such that $q < q'' < q'$. Thus from the definition of ρ, we have $(p', q') = \rho\ (A \not\rhd B, q)$.

8) $\rho\ (A \not\rhd B, k)$ (Figure 4.4):

Since $A \not\rhd B \subseteq A$, $\rho\ (A \not\rhd B, k) = \rho\ (A, k)$ if and only if $\nexists\ (p', q') \in B$ such that $(p', q') \sqsubseteq \rho\ (A, k)$. Following the equation, let $(p, q) = \rho\ (A, k)$ and let $(p', q') = \tau\ (B, p)$. By the definition of τ, we have $p' \geq p$. Thus, if $q' \leq q$ then $(p', q') \sqsubseteq (p, q)$ and $\rho\ (A \not\rhd B, k) \neq \rho\ (A, k)$. Indeed $\forall\ (p'', q'') \in A$ where $p < p'' \leq p'$ and $q' \leq q < q''$, we have $(p'', q'') \neq \rho\ (A \not\lhd B, k)$. Therefore, and from the definition of ϵ, $\rho\ (A \not\rhd B, k) = \tau\ (A \not\rhd B, p' + \epsilon)$.

Now, suppose $q' > q$. Assume $\exists\ (p'', q'') \in B$ such that $(p'', q'') \sqsubseteq (p, q)$, implying that $q'' < q'$. By the definition of τ, if $p'' \geq p$ then $p'' > p'$. Therefore $(p'', q'') \sqsubset (p', q')$, contradicting for B the defining property of GC-lists.

9) $\tau\ (A \triangle B, k)$ (Figure 4.5):

Following the equation, let $(p, q) = \tau\ (A, k)$, $(p', q') = \tau\ (B, k)$, $(p'', q'') = \tau'\ (A, \max(q, q'))$, and $(p''', q''') = \tau'\ (B, \max(q, q'))$. In addition, let $r = \min(p'', p''')$ and $s = \max(q'', q''')$. Now, either i) $q = \max(q, q')$ and therefore, from the definition of τ', we have $(p'', q'') = (p, q)$ and $q''' \leq q$, or ii) $q' = \max(q, q')$ and therefore we have $(p''', q''') = (p', q')$ and $q'' \leq q'$. In either case, $\max(q, q') = \max(q'', q''') = s$.

We seek to show a) that $(r, s) \in A \triangle B$, and b) that $\nexists (r', s') \in A \triangle B$ such that $k \leq r' < r$ and therefore that $\tau\ (A \triangle B, k) = (r, s)$.

a) First, we note that $(p'', q'') \in A$, $(p'', q'') \sqsubseteq (r, s)$, $(p''', q''') \in B$, and $(p''', q''') \sqsubseteq (r, s)$. Now, assume $\exists (r', s') \in A \triangle B$ such that $(r', s') \neq (r, s)$ and $(r', s') \sqsubset (r, s)$. Now, either $r' > r$ (and $s' \leq s$) or $s' < s$ (and $r' \geq r$). There are now four cases:

   i) $r' > r$ and $p'' \leq p'''$:
   In this case $\exists (u, v) \in A$ such that $u > p''$ and $v \leq \max(q, q')$ implying $(p'', q'') \neq \tau'\ (A, \max(q, q'))$.

ii) $r' > r$ and $p'' \geq p'''$:
In this case $\exists (u', v') \in B$ such that $u' > p'''$ and $v' \leq \max(q, q')$ implying $(p''', q''') \neq \tau' (B, \max(q, q'))$.

iii) $s' < s$ and $q'' \geq q'''$:
In this case $\exists (u, v) \in A$ such that $(u, v) < (p'', q'') = (p, q)$ with $u > r \geq k$, implying that $(p, q) \neq \tau (A, k)$.

iv) $s' < s$ and $q'' \leq q'''$:
In this case $\exists (u', v') \in B$ such that $(u', v') < (p''', q''') = (p', q')$ with $u' > r \geq k$, implying that $(p', q') \neq \tau (B, k)$.

Since all four cases lead to a contradiction, $(r, s) \in A \triangle B$.

b) Assume $\exists (r', s') \in A \triangle B$ such that $k \leq r' < r$. Since $A \triangle B$ is a GC-list $s' < s$. From the definition of $A \triangle B$, $\exists (u, v) \in A$ and $\exists (u', v') \in B$ such that $(u, v) \sqsubseteq (r', s')$ and $(u', v') \sqsubseteq (r', s')$. However, if $q = \max(q, q') = \max(q'', q''') = s$ then $(u, v) = \tau (A, k) \neq (p, q)$, a contradiction, or if $q' = \max(q, q') = \max(q'', q''') = s$ then $(u', v') = \tau (B, k) \neq (p', q')$, a contradiction.

Therefore $\tau (A \triangle B, k) = (r, s)$.

10) $\rho (A \triangle B, k)$ (Figure 4.5):

We prove a more general equation

$$
\begin{array}{l}
\rho (Q, k) = \\
\quad \textbf{let} \\
\quad\quad (p, q) = \tau' (Q, k - \epsilon) \\
\quad \textbf{in} \\
\quad\quad \tau (Q, p + \epsilon)
\end{array}
$$

Proof of this equation also establishes the equations for $\rho (A \bigtriangledown B, k)$, and $\rho (A \Diamond B, k)$.

Let $(p, q) = \tau' (Q, k - \epsilon)$. By the definition of τ', $q \leq k - \epsilon$ and $\nexists (p'', q'') \in Q$ such that $q < q'' \leq k - \epsilon$. By the definition of ϵ, we have $q < k$ and $\nexists (p'', q'') \in Q$ such that $q < q'' < k$. Now, let $(p', q') = \tau (Q, p + \epsilon)$. From the definitions of τ and ϵ, $p' > p$ and $\nexists (p''', q''') \in Q$ such that $p < p''' < p'$. Since Q is a GC-list and since $\nexists (p'', q'') \in Q$ such that $q < q'' < k$, we have $q' \geq k$. Assume $\exists (q''', p''') \in Q$ such that $k \leq q''' < q'$. Since $q < k$, we have $q < q''' < q'$.

Since Q is a GC-list, $q < q''' < q'$ implies that $p < p''' < p'$, a contradiction. No such (q''', p''') can exist. Since $q' \geq k$ and $\nexists(q''', p''') \in Q$ such that $k \leq q''' < q'$ we have $\rho\,(Q, k) = (p', q')$ as required.

11) $\tau\,(A \bigtriangledown B, k)$ (Figure 4.5):

Following the equation, let $(p, q) = \tau\,(A, k)$ and let $(p', q') = \tau\,(B, k)$.

We begin by showing that either $\tau\,(A \bigtriangledown B, k) = (p, q)$ or $\tau\,(A \bigtriangledown B, k) = (p', q')$, and following show that the conditional expression correctly selects from these two possibilities.

Assume $(u, v) = \tau\,(A \bigtriangledown B, k)$, with both $(p, q) \neq (u, v)$ and $(p', q') \neq (u, v)$. By theorem 4.7 we have $A \bigtriangledown B \subseteq A \cup B$ and therefore $(u, v) \in A \cup B$. There are now two cases:

a) $(u, v) \in A$

By the definition of τ : $u \geq k$, $p \geq k$ and $\nexists(p'', q'') \in A$ such that $k \leq p'' < p$. Therefore, $k \leq p < u$. Further, and also by the definition of τ , $\nexists(u', v') \in A \bigtriangledown B$ such that $k \leq u' < u$. Therefore, $(p, q) \notin A \bigtriangledown B$.

Since $(p, q) \in A$ but $(p, q) \notin A \bigtriangledown B$, $\exists(p''', q''') \in A \bigtriangledown B$ such that $(p''', q''') \sqsubset (p, q)$ (theorem 3.3 and theorem 4.9). Thus $(p''', q''') \notin A$, for otherwise A would not be a GC-list. Therefore, $(p''', q''') \in B$.

Since $(p''', q''') \sqsubset (p, q)$, we have $p''' \geq p$ and $q''' \leq q$. Since $(p, q) \in A$ and $(u, v) \in A$ and $p < u$, we have $q < v$ and therefore $q''' < v$. Since $(p''', q''') \in A \bigtriangledown B$ and $(u, v) \in A \bigtriangledown B$ and $q''' < v$, we have $p''' < u$. Since $p \geq k$ and $p''' \geq p$, we have $p''' \geq k$. But by the definition of τ , $\nexists(p''', q''') \in A \bigtriangledown B$ such $k \leq p''' < u$, a contradiction.

Therefore, $(u, v) \notin A$.

b) $(u, v) \in B$.

By the definition of τ : $u \geq k$, $p' \geq k$ and $\nexists(p''', q''') \in B$ such that $k \leq p''' < p'$. Therefore, $k \leq p' < u$. Further, and also by the definition of τ , $\nexists(u', v') \in A \bigtriangledown B$ such that $k \leq u' < u$. Therefore, $(p', q') \notin A \bigtriangledown B$.

Since $(p', q') \in B$ but $(p', q') \notin A \bigtriangledown B$, $\exists(p'', q'') \in A \bigtriangledown B$ such that $(p'', q'') \sqsubset (p', q')$ (theorem 3.3 and theorem 4.9). Thus $(p'', q'') \notin B$, for otherwise B would not be a GC-list. Therefore, $(p'', q'') \in A$.

Since $(p'', q'') \sqsubset (p', q')$, we have $p'' \geq p'$ and $q'' \leq q'$. Since $(p', q') \in B$ and $(u, v) \in B$ and $p' < u$, we have $q' < v$ and therefore $q'' < v$. Since $(p'', q'') \in A \bigtriangledown B$ and $(u, v) \in$

$A \bigtriangledown B$ and $q'' < v$, we have $p'' < u$. Since $p' \geq k$ and $p'' \geq p'$, we have $p'' \geq k$. But by the definition of τ, $\nexists (p'', q'') \in A \bigtriangledown B$ such $k \leq p'' < u$, a contradiction.

Therefore, $(u, v) \notin B$.

Since $(u, v) \notin A$ and $(u, v) \notin B$, we have $(u, v) \notin A \cup B$, a contradiction. Therefore $\tau\,(A \bigtriangledown B, k) = (p, q)$ or $\tau\,(A \bigtriangledown B, k) = (p', q')$.

Returning to the equation, the conditional expression in the body of the equation determines which of (p, q) and (p', q') is $\tau\,(A \bigtriangledown B, k)$. There are three cases, one corresponding to the first branch of the conditional expression $(q < q')$, one corresponding to the second branch of the conditional expression $(q > q')$, and one corresponding to the third branch of the conditional expression $(q = q')$.

a) $q < q'$:
If $p \geq p'$ then $(p, q) \sqsubset (p', q')$, and therefore $(p', q') \notin A \bigtriangledown B$ and $\tau\,(A \bigtriangledown B, k) = (p, q)$.
If $p < p'$ and $(p', q') \in A \bigtriangledown B$ we have $(p, q) < (p', q')$ implying $\tau\,(A \bigtriangledown B, k) \neq (p', q')$ and therefore that $\tau\,(A \bigtriangledown B, k) = (p, q)$.

b) $q > q'$:
If $p \leq p'$ then $(p', q') \sqsubset (p, q)$ and therefore $(p, q) \notin A \bigtriangledown B$ and $\tau\,(A \bigtriangledown B, k) = (p', q')$.
If $p > p'$ and $(p, q) \in A \bigtriangledown B$ we have $(p', q') < (p, q)$ implying $\tau\,(A \bigtriangledown B, k) \neq (p, q)$ and therefore that $\tau\,(A \bigtriangledown B, k) = (p', q')$.

c) $q = q'$:
If $p > p'$ then $(p, q) \sqsubset (p', q')$, implying $(p', q') \notin A \bigtriangledown B$ and $\tau\,(A \bigtriangledown B, k) = (p, q) = (\max(p, p'), q)$. If $p < p'$ then $(p', q') \sqsubset (p, q)$, implying $(p, q) \notin A \bigtriangledown B$ and $\tau\,(A \bigtriangledown B, k) = (p', q') = (\max(p, p'), q)$. If $p = p'$ then $\tau\,(A \bigtriangledown B, k) = (p, q) = (p', q') = (\max(p, p'), q)$.

12) $\rho\,(A \bigtriangledown B, k)$ (Figure 4.5):

Proof of this equation follows from the proof of the equation for $\rho\,(A \bigtriangleup B, k)$.

13) $\tau\,(A \Diamond B, k)$ (Figure 4.6):

Following the equation, let $(p, q) = \tau\,(A, k)$, $(p', q') = \tau\,(B, q + \epsilon)$ and $(p'', q'') = \tau'\,(A, p' - \epsilon)$. We seek to show a) that $(p'', q') \in A \Diamond B$, and b) that $\nexists (r, s) \in A \Diamond B$ such that $k \leq r < p''$ and therefore that $\tau\,(A \Diamond B, k) = (p'', q')$.

a) From the definition of τ, and from the definition of ϵ, we have $q < p'$. From the definition of τ', and from the definition of ϵ, we have $q'' < p'$. Since $(p'', q'') \in A$ and $(p', q') \in B$ and $q'' < p'$, we have $(p'', q') \notin A \Diamond B$ if and only if $\exists (r, s) \in A \Diamond B$ such that $(r, s) \sqsubset (p'', q')$. If $(r, s) \in A \Diamond B$ then $\exists (r'', s'') \in A$ and $\exists (r', s') \in B$ such that $r \leq r'' \leq s'' < r' \leq s' \leq s$. Now, either $r > p''$ (and $s \leq q'$), or $s < q'$ (and $r \geq p''$), giving two cases:

   i) $r > p''$ and $s \leq q'$:
   Since $r > p''$ we have $r'' > p''$. If $s'' \leq q''$ then $(r', s') \sqsubset (p'', q'')$ and A is not a GC-list. If $s'' > q''$ and $s'' \geq p'$ then $r' > p'$, implying $(r', s') \sqsubset (p', q')$ and B is not a GC-list. If $s'' > q''$ and $s'' < p'$ then $(p'', q'') \neq \tau' (A, p' - \epsilon)$.

   ii) $r \geq p''$ and $s < q'$:
   Since $s < q'$ we have $s' < q'$. If $r' \geq p'$ then $(r', s') \sqsubset (p', q')$ and B is not a GC-list. If $r' < p'$ and $r' \leq q''$ then $s'' < q''$, implying $(r'', s'') \sqsubset (p'', q'')$ and A is not a GC-list. From the definition of τ and τ' we have $q'' \geq q$. If $r' < p'$ and $r' > q''$ then $s' < q'$ and $(p', q') \neq \tau (B, q + \epsilon)$.

   Since both cases lead to a contradiction, we have $(p'', q') \in A \Diamond B$.

b) Assume $\exists (r, s) \in A \Diamond B$ such that $k \leq r < p''$. Since $(r, s) \in A \Diamond B$, $\exists (r'', s'') \in A$ and $\exists (r', s') \in B$ such that $s'' < r'$ and $(r'', s') \sqsubseteq (r, s)$. Indeed, $(r'', s') = (r, s)$, for otherwise $A \Diamond B$ would not be a GC-list. Now, since $(r', s') < (p', q')$ we have $r' \leq q$, for otherwise $(p', q') \neq \tau (B, q + \epsilon)$. Since $r' \leq q$ and $s'' < r'$, we have $k \leq r'' < p$ and $(p, q) \neq \tau (A, k)$, a contradiction.

Therefore $\tau (A \Diamond B, k) = (p'', q'')$.

14) $\rho (A \Diamond B, k)$ (Figure 4.6):

Proof of this equation follows from the proof of the equation for $\rho (A \triangle B, k)$.

□

4.6 Implementation

We examine implementation of the algebra from a theoretical perspective and from an applied perspective. We examine both the time required to find all solutions to a query and the expected

$\mathcal{P}(Q) =$
  $(u, v) \leftarrow \tau\ (Q, -\infty)$;
  while $u \neq \infty$ **do begin**
    Output (u, v);
    $(u, v) \leftarrow \tau\ (Q, u + \epsilon)$;
  end;

Figure 4.14: Driver procedure.

time to find a single solution to a query. This distinction is important, since a user might often require only a small number of solutions before modifying the query or finding the information she requires.

The algebra has been implemented and used in practice. We describe a database system that uses the algebra as an intermediate language. As a specific example, we discuss the use of the database system to implement a Network News server. An additional application of the database system is described as part of Chapter 6.

4.6.1 Efficiency

The access function equations defined in Section 4.4 may be interpreted operationally as recursive functions expressed in a functional programming language. The driver procedure of Figure 4.14 may be used to report all solutions to a query Q in increasing order. The procedure uses iterative calls to τ to generate the solutions. An equivalent driver procedure may be written using ρ. Corresponding driver procedures using either τ' or ρ' generate the solutions in reverse order. Required storage depends only on the size of the query, and not on the size of the database.

Consider the number of calls to access functions for sub-expressions (sub-queries) made during the evaluation of a single call to one of the access functions. For the combination, ordering and projection operators a fixed number of calls are made: four in the case of the "both of" combination operator ($\triangle$), two in the case of the "one of" combination operator ($\bigtriangledown$), three in the case of the ordering operator ($\Diamond$), and a single call for either of the projection operators (π_1 and π_2). For fixed-length intervals, an access function call is completed in constant time. For the index function over a particular symbol, an access function call is completed in $O(\log m)$ time, where m is the length of the postings list for that symbol (Section 4.4.1). These observations are combined in the following theorem which considers the evaluation of an access function for a query in which the containment operators are not used.

Theorem 4.14
Let Q be a query that does not involve a containment operation. Let m be the length of the longest postings list for a symbol referenced by Q. The time to compute a call to an access function for Q is $O(\log m)$ in the worst case.

Proof:

The proof is a simple induction over the height of an expression tree representing the query. If the tree is height one, the query is a fixed-length interval expression or a reference to the index function. An access function call for a fixed-length interval expression requires constant time; an access function call for the index function requires $O(\log m)$ time. Now, assume that the height of the expression tree for Q is greater than one, and that the theorem holds for the sub-queries (or sub-query) of Q. Since Q represents a combination, ordering or projection of its operand(s), the query may be solved with a constant number of calls to access functions for the operand(s), each of which requires $O(\log m)$ time in the worst case. □

The access functions for the containment operators make a single call to the access functions for each of their operands and then check that the containment relationship is satisfied. If the containment check fails, a (tail) recursive call is made, bypassing infeasible candidates. The time required to evaluate a call to an access function for a containment operator depends on the number of recursive calls that must be made before a solution is found. In the worst case, all elements of the operands may have to be eliminated, and a solution may not be found at all.

Since a bound cannot be placed on the work required to evaluate a single access function for a containment operation, we instead examine the behaviour of the containment operators when all solutions are generated using the driver procedure of Figure 4.14. Of particular interest, are the effects of indexing into the GC-lists.

Theorem 4.15
Let $Q = A \circ B$, where $\circ$ is one of the containment operators: $\lhd$, $\rhd$, $\not\lhd$ or $\not\rhd$. While generating all solutions to Q, the driver procedure $\mathcal{P}(Q)$ makes $O(|Q| + \min(|A|, |B|))$ calls to access functions for A and B.

Proof:

The essence of the argument is as follows: Each unsuccessful call to τ for $A \circ B$, which leads to a further tail-recursive call, eliminates from further consideration an element of A and an

element of B, giving rise to the $\min(|A|, |B|)$ term. Each successful call to τ for $A \circ B$ generates a new solution, giving rise to the $|Q|$ term.

We examine in detail the case for $Q = A \lhd B$. The detailed argument for the remaining operators is similar.

Let $a_0, a_1, ..., a_{|A|+1}$ be the sequence of elements of A, and let $b_0, b_1, ..., b_{|B|+1}$ be the sequence of elements of B, with $a_0 = b_0 = (-\infty, -\infty)$ and $a_{|A|+1} = b_{|B|+1} = (\infty, \infty)$. To establish an invariant over the **while** loop of Figure 4.14, we define i and j such that $a_i = (u, v)$ and $b_j = \rho(B, u)$ after $\mathcal{P}$ outputs (u, v). At the beginning of the procedure we assume $i = j = 0$, so that $a_i = b_j = (u, v) = (-\infty, -\infty)$. At the end of the execution, $i = |A| + 1$, $j = |B| + 1$ and $a_i = b_j = (u, v) = (\infty, \infty)$. Each iteration of the **while** loop increases the value of i by at least one, generating a new solution with each iteration.

We now establish an invariant within the body of τ for $A \lhd B$. Before the conditional expression is checked, we define i' and j' such that $a_{i'} = \tau(A, k) = (p, q)$ and $b_{j'} = \rho(B, q) = (p', q')$, with $k \geq u'$, $i' > i$ and $j' \geq j$. On a call from within $\mathcal{P}$, the invariant holds since $a_i \sqsubseteq b_i$ and $k = u + \epsilon$. If $p' \leq p$ then $a_{i'} \sqsubseteq b_{j'}$, the call succeeds, and $a_{i'}$ is returned as a solution. When $a_{i'}$ is output by $\mathcal{P}$, i is advanced to i'.

Otherwise, if $p' > p$ then $j' > j$. When the recursive call is made, we advance j to j', re-establishing the invariant. Each unsuccessful call to τ for $A \lhd B$ advances i' by one and advances j' by one. Since i' is bounded by $|A| + 1$ and j' is bounded by $|B| + 1$, the total number of unsuccessful calls is bounded by $\min(|A|, |B|) + 1$.

□

In the worst case, a call to an access function for a containment operation depends on the size of the smallest of its operands. If the size of the database is increased through the addition of new text, the size of each operand may increase proportionally without a corresponding increase in the size of the solution set for the overall containment relationship, consequently increasing the time required to find a solution. On the other hand, we might assume that the text is produced by a source with fixed statistical characteristics. In this case, the size of the operand sets and the result set may be expected to increase proportionally as text is added to the database. Under this assumption, we examine the expected number of calls to access functions for operands during the computation of an access function call for a containment operation. The number of calls to access functions for the operands will depend on probabilities associated with the containment relationships between them.

For an expression $A \circ B$, where $\circ$ is any of the containment operators ($\lhd$, $\rhd$, $\not\lhd$, or $\not\rhd$), we define

$$\Pr[A \circ B]$$

as the probability that an element of A satisfies the containment condition. We further assume that this probability is constant for the particular sub-queries A and B over text generated by the source. For a particular database generated by the source we have:

$$\Pr[A \circ B] \approx \frac{|A \circ B|}{|A|}$$

Consider evaluation of the access function $\tau\,(A \lhd B, k)$ from Figure 4.4. Initial calls are made to access functions for A and B.

$$\begin{aligned} (p, q) &= \tau\,(A, k) \\ (p', q') &= \rho\,(B, q) \end{aligned}$$

If a solution is not found, the access function recursively calls itself

$$\tau\,(A \lhd B, q')$$

where we note that q' is the end position of an element of B.

Consider now the evaluation of this recursive call. The initial step is again to call access functions for A and B

$$\begin{aligned} (p_1, q_1) &= \tau\,(A, q') \\ (p_1{}', q_1{}') &= \rho\,(B, q_1) \end{aligned}$$

If $(p', q') \in B \rhd A$ then (p_1, q_1) is the first element of A contained in (p', q'). By our assumptions, the probability that $(p', q') \in B \rhd A$ is $\Pr[B \rhd A]$. If $(p_1, q_1) \in A \lhd B$ then $(p_1{}', q_1{}') = (p', q')$ and (p_1, q_1) is returned as the solution. Otherwise, by our assumptions, the probability that $(p_1, q_1) \in A \lhd B$ is $\Pr[A \rhd B]$. If $(p_1, q_1) \in A \lhd B$ then $(p_1{}', q_1{}')$ is the first element of B that contains (p_1, q_1). If $(p_1{}', q_1{}')$ does not contain (p_1, q_1) then a recursive call will again be made. The probability that this recursive call will not be made, and that a solution will be returned is

therefore

$$\begin{aligned} P_{A \lhd B} &= \Pr[B \rhd A] + \Pr[A \lhd B](1 - \Pr[B \rhd A]) \\ &= \Pr[B \rhd A] + \Pr[A \lhd B]\ \Pr[B \not\rhd A] \end{aligned}$$

The expected number of calls before a solution is found is

$$\frac{1}{P_{A \lhd B}}$$

Note that if A and B are hierarchically related, with either $\Pr[A \lhd B] = 1$ or $\Pr[B \rhd A] = 1$, we have $P_{A \lhd B} = 1$, and solutions are found after a constant number of calls to access functions for the sub-queries. Further, $\Pr[A \lhd B] = 0$ if and only if $\Pr[B \rhd A] = 0$.

The access function $\rho(A \lhd B, k)$ makes a single call to an access function for A followed by a call to the access function τ for $A \lhd B$. This call in turn may make recursive calls to itself. As we have seen, the probability of one of these recursive calls successfully finding a solution without further recursion is $P_{A \lhd B}$.

Under a similar analysis, the access function calls $\tau'(A \lhd B, k)$ and $\rho'(A \lhd B, k)$ also result in recursive calls that are successful with probability $P_{A \lhd B}$.

We call the probability $P_{A \lhd B}$ the *containment probability* for $A \lhd B$. If $P_{A \lhd B} > 0$, we say that the query $A \lhd B$ has *positive containment probability*. Equivalent probabilities for the other containment operators are as follows:

$$\begin{aligned} P_{A \rhd B} &= \Pr[B \lhd A] + \Pr[A \rhd B]\ \Pr[B \not\lhd A] \\ P_{A \not\rhd B} &= \Pr[A \not\rhd B] \\ P_{A \not\lhd B} &= \Pr[A \not\lhd B] \end{aligned}$$

The containment probability $P_{A \not\lhd B}$ does not depend on $\Pr[B \not\rhd A]$, nor does $P_{A \not\rhd B}$ depend on $\Pr[B \not\lhd A]$. This is in contrast to $P_{A \lhd B}$ and $P_{A \rhd B}$: $P_{A \lhd B}$ depends on both $\Pr[A \lhd B]$ and $\Pr[B \rhd A]$; $P_{A \rhd B}$ depends on both $\Pr[A \rhd B]$ and $\Pr[B \lhd A]$. This difference is a consequence of the following observation: If we sample B giving (u, v), then the single access function call $\tau(A, u)$ will find an element of $A \lhd B$ with probability $\Pr[B \rhd A]$. Similarly, the single access function call $\rho(A, v)$ will find an element of $A \rhd B$ with probability $\Pr[B \lhd A]$. However, no call to an access function for A can produce an element which we know

to be in $A \not\rhd B$ or $A \not\lhd B$. The access function can only directly confirm that the particular element of B is in $B \not\rhd A$ or $B \not\lhd A$.

We now direct the discussion toward the proof of the following theorem:

Theorem 4.16
Let Q be a query such that it and its sub-queries all have positive containment probability. Let m be the length of the longest postings list for a symbol referenced by Q. The expected time to compute a call to an access function for Q is $O(\log m)$.

Proof:

As for Theorem 4.14, the proof proceeds by induction over the height of an expression tree representing the query If the tree is height one, the query is a fixed-length interval expression or a reference to the index function. An access function call for a fixed-length interval expression requires constant time; an access function call for the index function requires $O(\log m)$ time.

Now, assume that the height of the expression tree for Q is greater than one, and that the theorem holds for the sub-queries (or sub-query) of Q. If Q represents a combination, ordering or projection of its operand(s), then the query may be solved with a constant number of calls to access functions for the operand(s). If Q expresses a containment relationship, since Q has positive containment probability, the query may be solved with a constant expected number of calls to access functions. In either case, the expected number of calls is constant. Since, by assumption, the expected time to compute each call is $O(\log m)$, the expected time to compute an access function call for Q is also $O(\log m)$.

□

If all containment probabilities are 1 we have a special case of the theorem, which generalizes Theorem 4.14:

Theorem 4.17
Let Q be a query such that it and its sub-queries all have containment probabilities 1. Let m be the length of the longest postings list for a symbol referenced by Q. The time to compute a call to an access function for Q is $O(\log m)$ in the worst case.

4.6.2 Experience

The structural query algebra described in this chapter has been used as the intermediate query language for the MultiText distributed text database system developed at the University of Waterloo [19, 77]. The primary focus of the MultiText project is on research issues arising from the construction of a scalable document repository. One of these research issues is the need to support a variety of document formats, while allowing queries to reference structure across the various formats, independent of schema. The structure algebra fills this role.

The architecture of the MultiText database system is shown in Figure 4.15. The architecture consists of a number of search systems (or *index engines*) and retrieval systems (*text servers*) connected by a network. Queries, expressed in the structure algebra, are dispatched to the index engines, which return the extents that satisfy the query. The design of these index engines provide both an efficient implementation of inverted lists for the query algebra, and a fault-tolerant data organization strategy that supports update of the inverted lists [26]. The solution extents returned from the index engines are forwarded to the appropriate text servers, which yield the final results. To preserve the abstraction of a single database, the burden of maintaining search engines, dispatching queries and marshalling results must not fall to the client. The architecture therefore includes a *marshaller/dispatcher* to automate this process.

From the client's viewpoint, the marshaller/dispatcher appears to be a single search engine. The marshaller/dispatcher maintains a list of available search engines, selects the set necessary to solve the query, and dispatches the query to each of them. It receives the various results, combines them and returns them to the user as if they were the result of a single search. The marshaller/dispatcher has the further responsibility of system maintenance, particularly for the application of updates.

As a prototype demonstration, the MultiText database system has been used to implement a server for USENET network news. The NetNews server maintains a collection of current network news articles, updated on an on-going basis. The server supports two client protocols: the Network News Transfer Protocol (NNTP) [63], and the Hyper Text Transfer Protocol (HTTP), in combination with the Hyper Text Markup Language (HTML). The NNTP protocol is used by standard news reading software for accessing network news repositories. While NNTP does not directly provide a search interface, NNTP operations are internally translated into queries by the NNTP interface software.

The HTTP interface provides full search and display capabilities for World-Wide-Web browsing clients. Users enter queries using HTML forms that allow them to directly reference the

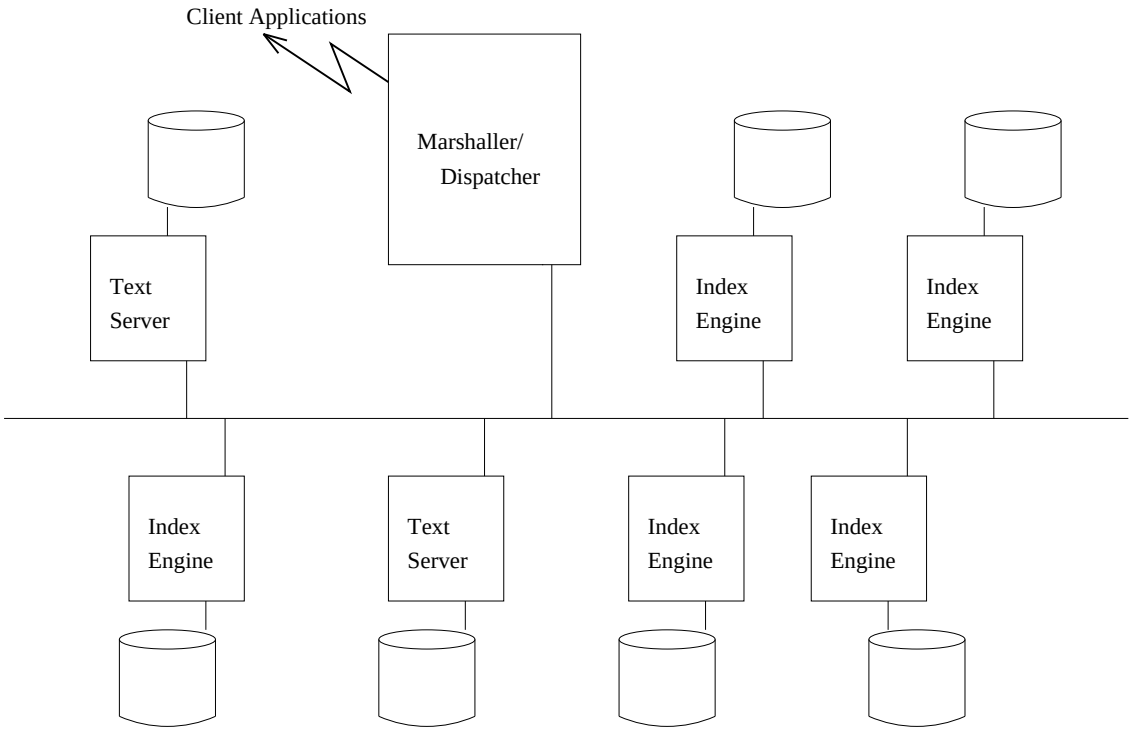


Figure 4.15: Architecture of the MultiText System.

structural elements of network news articles (header, subject, author, newsgroup etc.). Articles matching a user's query are displayed in a compact format, consisting of the article's subject, author and newgroup information, and including sample solutions to the user's query in context, if this is appropriate. The structure algebra is used to query directly for this information, so that entire articles need not be retrieved in order to display the context of the solution. Hyperlinks are provided to display the full article, to search for articles by the same author, to search for articles on the same subject, and to search for related articles in the same conversational thread. These links are implemented as further queries into the news database.

4.6.3 Memoization

As an efficiency improvement, the MultiText implementation of the structure algebra includes memoization of the most recent solutions to queries and their sub-queries. For each query Q, the implementation maintains information associated with the last call to each access function for Q. We represent this information in the form of tuples with default values as follows:

$$\begin{aligned}
(k_{\tau}, p_{\tau}, q_{\tau}) &= (\infty, \infty, \infty) \\
(k_{\rho}, p_{\rho}, q_{\rho}) &= (\infty, \infty, \infty) \\
(k_{\tau'}, p_{\tau'}, q_{\tau'}) &= (-\infty, -\infty, -\infty) \\
(k_{\rho'}, p_{\rho'}, q_{\rho'}) &= (-\infty, -\infty, -\infty)
\end{aligned}$$

At all times we have the following invariant over the values of the tuples:

$$\begin{aligned}
\tau(Q, k_{\tau}) &= (p_{\tau}, q_{\tau}) \\
\rho(Q, k_{\rho}) &= (p_{\rho}, q_{\rho}) \\
\tau'(Q, k_{\tau'}) &= (p_{\tau'}, q_{\tau'}) \\
\rho'(Q, k_{\rho'}) &= (p_{\rho'}, q_{\rho'})
\end{aligned}$$

On the return from a call to an access function for Q the value of the appropriate tuple is updated.

This memoization is of greatest value with queries of the form $Q = A \bigtriangledown B$, in which query A has a large number of solutions that may be found relatively quickly, and query B has a small number of solutions that require relatively more time. Generating all the solutions to Q is essentially a merge of A and B. The generation of each solution arising from A requires that we generate a solution from B for comparison purposes. Without memoization, the solution from B would be discarded; often, the same solution would then be re-generated for comparison with the next element of A. The memoization avoids this repeated effort.

4.7 Generalized Operators

The two combination operators may be generalized into a single operator with greater flexibility. Fixed-length intervals may be generalized into an enumeration operation that may be applied to any GC-list. Neither generalization introduces new semantic capabilities, but both have implementations that improve over a direct approach and both represent useful notation.

4.7.1 Combination

Each extent in the solution of

$$A_0 \triangle A_1 \triangle A_2 ... \triangle A_{m-1}$$

contains an extent from all of $A_0 ... A_{m-1}$. Each extent in the solution of

$$A_0 \triangledown A_1 \triangledown A_2 ... \triangledown A_{m-1}$$

consists of an extent from one of $A_0 ... A_{m-1}$. These two expressions represent extremes of a more general operation: the combination of n extents from m GC-lists. The combination operators may be used in concert to build these combinations. For example, each extent in

$$(A \triangle (B \triangledown C)) \triangledown (B \triangle (A \triangledown C)) \triangledown (C \triangle (A \triangledown B))$$

contains an extents from two of the three GC-lists: A, B and C. Unfortunately, following this pattern, an expression for combining n extents from m GC-lists has size

$$m \left(\begin{array}{c} m \\ n-1 \end{array} \right)$$

Nonetheless, the operation has intuitive appeal and is of significant practical use. A common situation in which this operation is of particular use is in selecting documents that contain a few of a large number of terms. During the early stages of a search session this operation might assist in narrowing down a list of search terms to those that retrieve the most relevant documents. We extend the algebra with an "n of m" operator that has a direct implementation.

Formally, we define the "n of m" operator as follows:

$$n \triangle (A_0, ..., A_{m-1}) = \mathcal{G} \left(\left\{ c \;\middle|\; \left| \{A \,|\, A \in \{A_0, ..., A_{m-1}\} \text{ and } \exists a \in A \text{ such that } a \sqsubset c\} \right| \geq n \right\} \right)$$

Each extent in $n \triangle (A_0, ..., A_{m-1})$ contains an element from at least n of the $A_0, ..., A_{m-1}$.

Definitions for the access functions are generalizations of those for $\triangle$ and $\triangledown$ (Figure 4.16). For example, the equation for $\tau(n \triangle (A_0, A_1, ..., A_{m-1}),\ k)$ first evaluates $\tau(A_i, k)$ for each of the sub-queries $A_0, A_1, ..., A_{m-1}$. Then let q be the first end point covering at least n of the resultant extents and let $B_0, ..., B_{l-1}$ be those members of $A_0, A_1, ..., A_{m-1}$ that end at or before q. The expression $\tau'(B_j, q)$ is evaluated for each of the $B_0, ..., B_{l-1}$. The resulting extents span the solution of $\tau(n \triangle (A_0, A_1, ..., A_{m-1}),\ k)$. Then let p be the last start point covering at least n of these extents. The extent (p, q) is then $\tau(n \triangle (A_0, A_1, ..., A_{m-1}),\ k)$.

$\tau\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k) =$
let
$(p_i, q_i) = \ \tau\ (A_i, k) \quad (0 \leq i < m)$
$q \in \{q_i\}$ such that
$|\{q_i \mid q_i < q\}| < n$ and
$|\{q_i \mid q_i \leq q\}| \geq n$
$\{B_0, ..., B_{l-1}\} = \{A_i \mid q_i \leq q\}$
$(u_j, v_j) = \ \tau'\ (B_j, q) \quad (0 \leq j < l)$
$p \in \{u_j\}$ such that
$|\{u_j \mid u_j > p\}| < n$ and
$|\{u_j \mid u_j \geq p\}| \geq n$
in
$(p,\ q)$

$\rho\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k) =$
let
$(p, q) = \ \tau'\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k - \epsilon)$
in
$\tau\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ p + \epsilon)$

$\tau'\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k) =$
let
$(p_i, q_i) = \ \tau'\ (A_i, k) \quad (0 \leq i < m)$
$p \in \{p_i\}$ such that
$|\{p_i \mid p_i > p\}| < n$ and
$|\{p_i \mid p_i \geq p\}| \geq n$
$\{B_0, ..., B_{l-1}\} = \{A_i \mid p_i \geq p\}$
$(u_j, v_j) = \ \tau\ (B_j, p) \quad (0 \leq j < l)$
$q \in \{v_j\}$ such that
$|\{v_j \mid v_j < q\}| < n$ and
$|\{v_j \mid v_j \leq q\}| \geq n$
in
$(p,\ q)$

$\rho'\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k) =$
let
$(p, q) = \ \tau\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ k + \epsilon)$
in
$\tau'\ (n \triangle\ (A_0, A_1, ..., A_{m-1}),\ q - \epsilon)$

Figure 4.16: Implementation framework for the generalized combination operator.

4.7.2 Enumeration

A fixed length extent (a member of Σ^n) covers exactly n words. We generalize this idea into an enumeration operation (A^n), where each extent in the solution covers exactly n extents of a GC-list.

$$A^n = \mathcal{G}\left(\{c \mid \exists a_0, ..., a_{n-1} \in A \text{ where } a_0 \neq ... \neq a_{n-1} \text{ and } a_0 \sqsubseteq c, ..., a_{n-1} \sqsubseteq c\}\right)$$

As an example of an enumeration operation, consider the informal query

Find three lines that contain "Birnam" and "Dunsinane".

which may be expressed as

$$\left(\mathcal{I}(\text{“<line>”}) \lozenge \mathcal{I}(\text{“</line>”})\right)^3 \triangleright \left(\mathcal{I}(\text{“birnam”}) \triangle \mathcal{I}(\text{“dunsinane”})\right)$$

Enumeration may be implemented using the existing operators.

$$A^n \equiv \underbrace{\pi_1(A) \lozenge \pi_1(A) \lozenge ...}_{n-1} \lozenge A$$

The benefit of an explicit enumeration operator may be seen in the evaluation of the expression

$$A^2 \equiv \pi_1(A) \lozenge A$$

Evaluating this expression using the equation for τ in Figure 4.6, we note that if

$$\begin{aligned}(p, q) &= \tau\left(\pi_1(A), k\right) \\ (p', q') &= \tau\left(A, q + \epsilon\right) \\ (p'', q'') &= \tau'\left(\pi_1(A), p' - \epsilon\right)\end{aligned}$$

we have

$$\tau'\left(\pi_1(A), p' - \epsilon\right) = \tau\left(\pi_1(A), k\right)$$

and evaluation of the τ' operation is not needed. Definitions for the access functions for enumeration are given in Figure 4.17.

$\tau\ (A^n,\ k) =$
if $n = 1$
$\tau\ (A, k)$
else
let
$(p, q) = \tau\ (\pi_1(A), k)$
$(p', q') = \tau\ (A^{n-1}, q + \epsilon)$
in
if $q' = \infty$
(∞, ∞)
else
(p, q')

$\rho\ (A^n,\ k) =$
let
$(p, q) = \tau'\ (A^n,\ k - \epsilon)$
in
$\tau\ (A^n,\ p + \epsilon)$

$\tau'\ (A^n,\ k) =$
if $n = 1$
$\tau'\ (A, k)$
else
let
$(p, q) = \tau'\ (\pi_2(A), k)$
$(p', q') = \tau'\ (A^{n-1}, p - \epsilon)$
in
if $p' = -\infty$
$(-\infty, -\infty)$
else
(p', q)

$\rho'\ (A^n,\ k) =$
let
$(p, q) = \tau\ (A^n,\ k + \epsilon)$
in
$\tau'\ (A^n,\ q - \epsilon)$

Figure 4.17: Implementation framework for enumeration.

4.8 Comparison with Other Work

The structure algebra described in the present chapter owes much of its intellectual and cultural heritage to two earlier structured text retrieval languages developed at the University of Waterloo: the query algebra developed by Burkowski for Textriever (described in Section 2.3.1) and the PAT text search system developed by the Centre for the New Oxford English Dictionary (described in Section 2.3.2). This final section provides a direct comparison between those efforts and the structure algebra.

4.8.1 Textriever

The results discussed in this chapter grew directly from attempts to extend the Burkowski's algebra for use in the MultiText project. Particular effort was directed toward achieving independence from a pre-defined fixed schema at the database level, toward correcting the semantic problems, toward providing appropriate ordering and proximity operators, and toward developing an efficient implementation, free from the need for special purpose data structures.

The operators of Burkowski's algebra may all be expressed directly in the structure algebra. The table in Figure 4.18 provides details of the mapping. In the case of the intersection and difference operators, if the operands are concordance lists then the results are concordance lists, preserving the semantics of Burkowski's algebra. The union high ($\Uparrow$) operator does not have a direct equivalent in the structure algebra. For this operator the necessary containment relationships must be expressed through a combination of containment operators.

Beyond the semantic problems discussed in Section 2.3.1, which are addressed by the structure algebra, Burkowski's algebra contains no equivalent of the ordering operator, the "both of" ($\triangle$) combination operator, the generalized combination operator, fixed-length intervals (Σ^n), or the generalized enumeration operator.

4.8.2 PAT

The structural operators of PAT may be expressed in the structure algebra. Details of the mapping are given in Figure 4.19. The mapping assumes that the index function ($\mathcal{I}$) maps the text and markup symbols onto character positions rather than onto word positions. Under a mapping emulating PAT, markup would be considered as part of the text and the index function would

Burkowski's Operator	*Meaning*	*Equivalent*
$A \overset{\uparrow}{\cap} B$	Intersect high	$A \rhd B$
$A \overset{\downarrow}{\cap} B$	Intersect low	$A \lhd B$
$A \overset{\uparrow}{-} B$	Difference high	$A \not\rhd B$
$A \overset{\downarrow}{-} B$	Difference low	$A \not\lhd B$
$A \overset{\uparrow}{\cup} B$	Union high	$(A \not\lhd B) \bigtriangledown (B \not\lhd A) \bigtriangledown (A \rhd B) \bigtriangledown (B \rhd A)$
$A \overset{\downarrow}{\cup} B$	Union low	$A \bigtriangledown B$
$n \rhd A$	Extension right	$A \Diamond \Sigma^n$
$n \lhd A$	Extension left	$\Sigma^n \Diamond A$

Figure 4.18: Mapping of Burkowski's algebra.

map symbols from both the text and markup alphabets onto the range of character positions each occupies.

A match point set is represented as a GC-list having elements with equal start and end points. Since it does not appear possible to construct a region set with equal start and end points using PAT region definitions, match point sets are distinct in the mapping.

This distinction is used in the mapping for the PAT union operator ("+"). Unlike the other operators, the mapping for the union operator is conditional. The definition is conditioned on determining if either operand is a match point set and, when both operands are region sets, whether the sets contain overlapping regions. The predicate for this condition is expressed using the structure algebra to specify the terms. An expression of the form

$$e \rhd \Sigma^2$$

is equal to the empty GC-list if and only if e is a match point set, with all extents one character in length.

Using PAT we are restricted to finding lines that contain the start of a speech or speeches that contain the start of a line (page 21). Using the structure algebra we may search specifically for lines that contain complete speeches, speeches that contain complete lines, lines containing the start of a speech, lines containing the end of a speech, speeches that contain the start of a line,

PAT *Operator*	*Equivalent*
`docs` m_0`..`m_1	$\pi_1(m_0) \Diamond \pi_1(m_1)$
e `including.`n m	$e \rhd \pi_1(m)^n$
e `not including.`n m	$e \not\rhd \pi_1(m)^n$
e `^` m	$e \rhd (\pi_1(e) \rhd \pi_1(m))$
e `-` m	$e \not\rhd (\pi_1(e) \rhd \pi_1(m))$
e `fby.`n m	$e \rhd \pi_1((\pi_1(e) \Diamond \pi_1(m)) \lhd \Sigma^n)$
e `near.`n m	$e \rhd (\pi_1(e) \lhd ((\pi_1(e) \triangle \pi_1(m)) \lhd \Sigma^n))$
e `within` r	$e \rhd (\pi_1(e) \lhd r)$
e_0 `+` e_1	$\begin{cases} \pi_1(e_0) \bigtriangledown \pi_1(e_1) & \text{if } (e_0 \rhd \pi_1(e_1)) \bigtriangledown (e_1 \rhd \pi_1(e_0)) \neq \emptyset, \\ & \text{or } e_0 \neq \emptyset \text{ and } e_0 \rhd \Sigma^2 = \emptyset, \\ & \text{or } e_1 \neq \emptyset \text{ and } e_1 \rhd \Sigma^2 = \emptyset \\ e_0 \bigtriangledown e_1 & \text{otherwise} \end{cases}$

Figure 4.19: Mapping of PAT operators.

and speeches that contain the end of a line:

$$\text{LINES} \rhd \text{SPEECHES}$$
$$\text{SPEECHES} \rhd \text{LINES}$$
$$\text{LINES} \rhd \pi_1(\text{SPEECHES})$$
$$\text{LINES} \rhd \pi_2(\text{SPEECHES})$$
$$\text{SPEECHES} \rhd \pi_1(\text{LINES})$$
$$\text{SPEECHES} \rhd \pi_2(\text{LINES})$$

Like Burkowski's algebra, PAT has no equivalent of the "both of" combination operator, the generalized combination operator, fixed-length intervals, or the generalized enumeration operator. Neither Burkowski's algebra nor PAT can solve a query such as

Find three lines that contain "Birnam" and "Dunsinane".

as neither can express the answer, which may contain overlapping intervals. The commonality among these operators, shared to an extent by the union operator, is that a solution cannot be properly expressed without generalized concordance lists.

4.8.3 PAT trees vs. Inverted Lists

The PAT text searching system is intimately associated with its original implementation using a PAT tree or PAT array [45]. The character array representing the text is viewed as a set of *semi-infinite strings*, or *sistrings*. Each sistring consists of the text beginning at particular position in the character array and continuing to the right to the end of the array. Each sistring typically begins at the start of a word or markup symbol, but may begin at each character position, or even at each bit position.

A PAT tree is essentially a PATRICIA tree [76] over the sistrings of the text. A PAT array [43] has as each of its elements a start position for a sistring of the text, with the positions sorted lexicographically according to the values of the sistrings. PAT arrays were developed independently as *suffix arrays* by Manber and Myers [74]. PAT trees and PAT arrays have properties that allow efficient implementation of search operations that are not easily implemented using inverted lists [45]. PAT trees and PAT arrays are particularly adept at phrase searching and lexicographical range searching, and may additionally be used for efficient regular expression searching, and for

the determination of most-frequent and longest repetitions of text. This last ability is used to implement the `signif` and `lrep` PAT operators. The trade-off between PAT arrays and PAT trees one of storage vs. time: PAT arrays require less than 25% of the storage of a PAT tree [45], but suffer from a penalty factor of $O(\log n)$ for some operations, where n is the length of the text. Recently, Clark and Munro [25] have described a compact implementation of *suffix trees*, which are similar to PAT trees, that have storage requirements competitive with those of PAT arrays.

Both PAT trees and PAT arrays implement phrase searching and lexicographical range searching in $O(\log n)$ time. In the case of a PAT array, these operations are effected using binary searching of the array. Unfortunately, when the array is stored on disk, a binary search requires $O(\log n)$ disk read accesses. On a text the length of the *New Oxford English Dictionary* a phrase search may require as many as 50-60 disk accesses [100]. On the other hand, a phrase search using a suffix tree can require as few as 3-4 disk accesses on a similarly sized text [25].

PAT arrays and PAT trees cause implementation difficulties for the structural operations of PAT, since solutions are generated in lexicographical order rather than in positional order. Gonnet et al. [45] describe the implementation of the query

$$m_0 \texttt{ fby.}n \ m_1$$

where m_0 and m_1 are match point sets generated from PAT arrays or PAT trees. The sub-queries m_0 and m_1 are first solved, the smaller of m_0 and m_1 is then sorted, and each element of the larger set is filtered against this sorted set, a procedure that requires $O((|m_0| + |m_1|)\log(\min(|m_0|, |m_1|)))$ time in total. The equivalent query in the structure algebra may be solved in $O(S \log(\max(|m_0|, |m_1|)))$ time, where S is the number of solutions, or, noting that no two solutions may share a start or end position, in $O(\min(|m_0|, |m_1|)\log(\max(|m_0|, |m_1|)))$ time, assuming that m_0 and m_1 are directly indexed by the index function. More importantly, the PAT implementation requires $O(\min(|m_0|, |m_1|)\log(\min(|m_0|, |m_1|)))$ time to generate *any* solution, since the smaller set must first be sorted. However, the result of any PAT structural operation is a sorted set, and these sets may be combined without further sorting.

Chapter 5

Pattern Matching in Structured Text

We now turn from indexed text to un-indexed text, from the problem of searching in a structured text database to the problem of searching through an un-indexed flat text for a specified pattern, while taking account of structure within this text.

Regular expressions are widely regarded as a precise, succinct notation for specifying a text search, with a straightforward efficient implementation. Many people routinely use regular expressions to specify searches in text editors and with stand-alone search tools such as the Unix `grep` utility. A regular expression search is generally understood in terms of a recognition problem: "Does a given string of text match a particular pattern?" However, searching is a different problem: "Find the substrings of a text that match a particular pattern."

The most widely available regular expression search utility, the Unix `grep` command, imposes a very rigid structure on search and retrieval. Patterns must be contained on a single line, and only lines containing or not containing the pattern may be selected for retrieval. Several attempts have been made to loosen the restrictions. The `agrep` search program allows the user to define record delimiters for searching [106]. The structural regular expressions provided in the text editor `sam` allow the use of regular expressions under a "left-most longest match" rule to divide a file into records for searching [81, 82].

In this chapter we examine this current practice in light of the results of the previous chapter. After reviewing the basics of regular expressions, we develop a theory of regular expression search under the shortest substring rule. To verify the practicality of the technique, a regular expression search tool, `cgrep`, has been written. The structure, performance and utility of this tool is discussed. Finally, we examine implementation of the structure algebra of the previous chapter as a text scanning tool, complementing regular expression search.

5.1 Background

5.1.1 Regular Expressions

A regular expression r denotes a language $L(r)$, a set of strings composed of symbols from an alphabet Σ. Any regular language may be denoted by a regular expression built from five primitives defined as follows:

r	$L(r)$	
λ	$\{\text{“”}\}$	(empty string)
a	$\{\text{“a”}\}$	(alphabet symbol $a \in \Sigma$)
$r_1 \mid r_2$	$L(r_1) \cup L(r_2)$	(alternation)
$r_1 r_2$	$L(r_1) \circ L(r_2)$	(concatenation)
r^*	$L(r)^*$	(repetition)

where the concatenation of two languages $L_1 \circ L_2$ is defined as

$$L_1 \circ L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

and L^*, the closure of a language L, is defined as the smallest solution to

$$L^* = \{\text{“”}\} \cup (L \circ L^*)$$

While these five primitives are sufficient to describe any regular language, the five primitives alone do not yield a concise notation. For example, consider the operators $\&$, $-$, and $+$ defined as follows:

r	$L(r)$	
$r_1 \,\&\, r_2$	$L(r_1) \cap L(r_2)$	(inclusion)
$r_1 - r_2$	$L(r_1) \setminus L(r_2)$	(exclusion)
r^+	$L(r) \circ L(r)^*$	(repetition at least once)

Regular languages are known to be closed under intersection and set difference, yet the expressions $r_1 \,\&\, r_2$ and $r_1 - r_2$ are not easily represented using the five primitives, and the representation may be exponentially larger than r_1 and r_2. Even simpler constructions cause exponential growth in regular expressions: r^+ is trivially defined in terms of r^* yet an equivalent expression

using only the five primitives may be exponentially larger than one using $+$. Throughout the chapter, these and other operators are defined as necessary, to succinctly present the approach. As an immediate example, a regular expression denoting all of the symbols in the alphabet may be easily constructed using the alternation operation and the symbols in the alphabet, but it is far simpler to use Σ itself to denote this regular expression.

Regular expressions are easily re-cast in terms of the recognition problem: We say that a string x matches the regular expression r if $x \in L(r)$. For any regular expression, a finite automaton may be constructed that solves the recognition problem in $O(|x|)$ time where $|x|$ is the number of symbols in x. A single left-to-right scan is made over x. Storage requirements depend only on properties of r, not on the length of x. a speech by a witch that contains the words "Dunsinane" or "Birnam" matches the regular expression:

$$\texttt{<speaker>}\,\Sigma^*\,\texttt{Witch}\,\Sigma^*\,\texttt{</speaker>}\,\Sigma^*\,(\texttt{Birnam} \mid \texttt{Dunsinane})\,\Sigma^*\,\texttt{</speech>}$$

The basic results in the theory of regular languages and finite automata were developed in the 1950's and early 1960's. A standard account of these results and their development is given by Hopcroft and Ullman [57, Chapter 2]. This account includes an algorithm for the conversion of a regular expression to a nondeterministic finite automata (NFA) and a discussion of the closure properties of regular languages. This material (or the equivalent) is assumed as background for the rest of the chapter.

5.1.2 Pattern Matching with Regular Expressions

At its most general, the problem of scanning and searching a continuous stream of text with a regular expression may be characterized as follows: "Given a string x and a regular expression r, find all substrings of x that match r." As we discussed in Chapter 3, the cardinality of such a solution set may be quadratic in the length of x. Searching this universe of all substrings may yield a plethora of overlapping and nested results. For this reason, any attempt at general search will usually apply a restriction to find and report only some subset of these solutions. These arbitrary restrictions, which alter the semantics of the search, often appear simple, but are difficult to formalize, difficult to use precisely, or difficult to implement efficiently.

The most common restriction is the "left-most longest match" rule. This is the rule mandated by the POSIX standard [59] and used in many software tools [50]. The rule is carefully stated in the rationale to the POSIX standard:

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

Having said this, when performing a general search, rather than looking for a single match, we are left with the question of what is the next match? There are two obvious choices: 1) begin the search again after the first character of the match, or 2) begin the search again after the last character of the match.

The second of these choices is the one usually taken, as the first choice may result in a large number of nested solutions. We refer to the technique of successively applying the left-most longest-match rule, starting each time after the last character of the match, as "longest-match disjoint substring search". We say "disjoint" to indicate that the solutions may not overlap or nest. Using this rule, searching the *Macbeth* text file with the regular expression

$$\texttt{<speaker>}\,\Sigma^*\,\texttt{Witch}\,\Sigma^*\,\texttt{<\speaker>}\,\Sigma^*\,\texttt{<\speech>}$$

results in a single match, starting at the fifth line of Figure 2.1 and continuing to the end of the last speech in the play. This clearly is not the intended result of this search. Furthermore, it is not obvious how to amend the regular expression to yield the intended result.

Aho [2,3] gives a general review of algorithms for patterning matching in text, with particular emphasis on regular expression search and practical experience gained with tools developed in conjunction with the Unix system. He discusses three possibilities for the output of a text search:

1. If the target text is pre-divided into records, such as lines, the records containing the pattern may be reported.
2. The longest-match disjoint substring rule may be used.
3. A "yes" or "no" may be reported, indicating if the target text contains the pattern.

He assumes the last case for the purposes of his review.

For his PhD thesis, Baeza-Yates [8] formalized the problem as that of finding all m such that the text t matches the regular expression $\Sigma^m r \Sigma^*$. These values of m specify the locations in the text where a match begins.

In 1968, Thompson described an algorithm for using regular expressions to search text [99]. Thompson's algorithm reports each point in the target text where a match ends. Equivalently, the algorithm finds all m such that the text t matches the regular expression $\Sigma^* r \Sigma^m$. A number of variants of Thompson's algorithm are described by Aho, Hopcroft and Ullman [5]. The algorithm described in Section 5.3.2 is a extension of one of the variants proposed in their discussion.

In the same year that Thompson published his regular expression search algorithm, a group at MIT used regular languages for automatically constructing lexical analyzers [62]. This system used the longest-match rule to resolve simple cases of matching ambiguity and reported errors in others. Since then, regular expressions, interpreted under the longest-match rule, have been widely used to specify scanners for programming languages and other translators [6, 69]. Scanning is a different problem from searching, requiring a partitioning of the input into tokens. The leftmost longest match rule appears to be a natural choice for scanning. Consider the typical definition of an identifier in a programming language, as an alphabetic character followed by zero or more alphanumeric characters. This definition must be interpreted under a longest-match rule to be correct.

Much effort has been devoted to addressing special cases of regular expression search. Search algorithms have been developed for finding single keywords [17, 58, 65], sets of keywords [4], keywords separated by sequences of "don't cares" [1, 40, 73], and other simple patterns [10, 107]. A general algorithm for regular expression search in preprocessed text with average-case time complexity of $O(\sqrt{n})$ has been developed by Baeza-Yates and Gonnet [8, 9]. That algorithm reports only the start points of matches.

5.2 Shortest Substrings

If L is a language over an alphabet Σ we define the language $G(L)$ as follows:

> *The string $x \in L$ is an element of $G(L)$ if and only if $\nexists y \in L$ such that y is a proper substring of x.*

By *proper substring* we mean that y is a substring of x and $y \neq x$. If L contains the empty string then $G(L) = \{\text{“”}\}$. For the remainder of this chapter we assume that L does not contain the empty string.

Theorem 5.1
If L is a regular language then $G(L)$ is a regular language.

Proof: $G(L) = L \setminus ((L(\Sigma^+) \circ L \circ L(\Sigma^*)) \cup (L(\Sigma^*) \circ L \circ L(\Sigma^+)))$ which is regular by the various closure properties of regular languages. □

From the proof, we see that this theorem applies equally well to any class of languages closed under set difference, Kleene closure and concatenation.

We may interpret shortest substring searching in terms of matching elements from a language L. If $x = a_1 a_2 ... a_n$, where the a_i are symbols from Σ, we can represent the result of such a search as a set of ordered pairs of integers, each giving a start and end position in x corresponding to an element in the result. We use the notation $x[u, v]$ with $u \leq v$ to indicate the substring of x starting at a_u and ending at a_v. Searching is defined as follows:

$$\textit{If } x \in \Sigma^* \textit{ and } L \textit{ is a language over } \Sigma \textit{ then } \mathcal{G}(L, x) = \{(u, v) \mid x[u, v] \in G(L)\}.$$

For example,

$$\mathcal{G}(\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}, \text{“abracadabra”}) = \{(1, 2), (4, 5), (8, 9)\}$$

Note that, although the string “abrac” is a member of the regular language denoted by $\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}$, it is not a member of $G(\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c})$ and the pair $(1, 5)$ is not included in the set. By Theorem 3.1, the elements of $\mathcal{G}(L, x)$ are totally ordered. As a consequence of Theorem 3.2 we have $|\mathcal{G}(L, x)| \leq n$.

The members of L reported by a longest-match disjoint substring search are dependent on the text being searched. For example, a longest-match search for the regular expression $\mathtt{ab} \mid \mathtt{a}\Sigma^*\mathtt{c}$ in the string “ababab” results in three matches of the form “ab”. If we add a final “c” to the end of the string (making the string “abababc”) the result changes to a single match of the form $\mathtt{a}\Sigma^*\mathtt{c}$. The string still contains three matches of the form “ab”, but these are no longer reported. In general, if $z \in L$ appears in the target text it is not possible to determine if a longest-match substring search will find a particular occurrence of z without reference to the entire text being searched. In contrast, over any text a shortest-match search reports all occurrences of the members of L that are in $G(L)$ and no others.

We return briefly to the problem of recognition to demonstrate how the problem of recognizing an element of L can be reduced to searching for elements of $G(L)$. If we have an algorithm

that searches a text for elements of $G(L)$, then by the addition of start and end tokens, the same algorithm may be used to recognize members of L.

Theorem 5.2
If ^ *and* $ *are not symbols in* Σ *then* $G(\{\text{“}\,\hat{}\,\text{”}\} \circ L \circ \{\text{“}\,\$\,\text{”}\}) = \{\text{“}\,\hat{}\,\text{”}\} \circ L \circ \{\text{“}\,\$\,\text{”}\}$.

Proof: Assume there exists $x = \hat{}\,w\$$ and $y = \hat{}\,z\$$, elements of $\{\text{“}\,\hat{}\,\text{”}\} \circ L \circ \{\text{“}\,\$\,\text{”}\}$ such that y is a proper substring of x. Since y is a proper substring of x, y must either be a substring of $\hat{}\,w$, or a substring of $w\$$. The first case implies that $ appears in w; the second implies that ^ appears in w. □

This theorem provides a hint on how to use shortest substring search to find objects, such as identifiers, that have a longest-match component implicit in their definitions. Instead of directly using the definition given earlier, we might specify an identifier as a non-alphanumeric character, followed by an alphabetic character, followed by zero or more alphanumeric characters, followed by a non-alphanumeric character. In the search results, a surrounding pair of non-alphanumeric characters will be reported with each identifier.

5.3 Substring Search Algorithms

In this section we compare algorithms for longest- and shortest-match substring search. A string may be recognized as a member of a regular language by a single left-to-right scan with constant store. We argue informally that there is no longest-match search algorithm that shares this property. Any longest-match search algorithm using constant store can be forced to make multiple scans. In contrast, we present a simple algorithm for shortest-match substring search that makes a single left-to-right scan over the string and uses storage dependent only on the regular expression to be matched. In both cases we assume that matches are reported as they are discovered and consequently no storage is required for the matches.

5.3.1 The Complexity of Longest-Match Disjoint Substring Search

Suppose we have an algorithm that performs a longest-match search of a regular expression with a single scan of the target string using only constant store. Consider a longest-match search with the regular expression $\texttt{ab} \mid \texttt{a}\Sigma^*\texttt{c}$ on a string of the form $(\texttt{ab})^n\texttt{d}$ for some fixed but arbitrary n.

The string contains exactly n matches of the form "ab". Since "a" is the initial symbol in the string, the algorithm must make a full scan of the string, examining every character, to determine that this initial symbol is not part of a match of the form $\mathtt{a}\Sigma^*\mathtt{c}$. Since n may be arbitrarily large, no constant store may be used to maintain the potential matches that would be discovered while this determination is being made. It appears that our supposed algorithm cannot exist.

In this specific case, a longest-match search may be performed with two scans and constant store. In practice, longest-match search algorithms do make multiple scans of portions of their target strings, but this is not a serious problem as searches are generally restricted to a single line of text.

5.3.2 Shortest-Match Substring Search

We detail an algorithm for shortest-match substring search for members of the regular language L over a text $x = a_1 a_2 ... a_n$ of length n. We assume that a nondeterministic finite automata M has been constructed to recognize L (perhaps from a regular expression). Let $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a set of states, Σ is an alphabet of symbols, δ is a state transition function mapping each element of $Q \times \Sigma$ onto a subset of Q, q_0 is the start state, and F is a set of final states. For simplicity, we assume that M has no transitions on the empty string (λ-transitions).

For the purposes of the algorithm, we assume that states are designated by numbers in the range 1 to $|Q|$ where the start state q_0 is assigned 1. The algorithm appears in figure 5.1. The two integer arrays P and P' are indexed by state number with each element holding an index into x or the value 0. The symbols i, j, q and u designate integer variables, used as counters and indices.

Storage requirements (except for the string itself) depend only on $|Q|$, the number of states of M. The outermost loop at lines 3–20 makes a single left-to-right scan over x. Noting that the loop at lines 8–9 makes at most $|Q|$ iterations for each iteration of the outer loop at lines 7–9, it is apparent from the structure of the loops that the algorithm has worst-case time complexity of $O(|Q|^2 n)$. That the algorithm correctly performs a shortest-match substring search is the subject of the next theorem. As a final note, in an actual implementation of the algorithm, the arrays P and P' are more efficiently represented as lists of states and positions. States for which an array element would be 0 are omitted from this list.

```
1     for i ← 1 to |Q| do
2         P_i ← 0;
3     for i ← 1 to n do begin
4         P_1 ← i;
5         for j ← 1 to |Q| do
6             P'_j ← 0;
7         for j ← 1 to |Q| do
8             for q ∈ δ(j, a_i) do
9                 P'_q ← max(P'_q, P_j);
10        u ← 0;
11        for j ← 1 to |Q| do
12            if j ∈ F then u ← max(u, P'_j);
13        if u > 0 then begin
14            Output (u, i);
15            for j ← 1 to |Q| do
16                if P'_j ≤ u then P'_j ← 0;
17        end;
18        for j ← 1 to |Q| do
19            P_j ← P'_j;
20    end;
```

Figure 5.1: Shortest-match substring search algorithm.

Theorem 5.3
A pair (u, v) is output by the algorithm of figure 5.1 if and only if $(u, v) \in \mathcal{G}(L, x)$.

Proof: We begin by establishing invariants for the array P over the loop at lines 3–20. At any point in the execution of the algorithm, let t be the first element of the previous pair output or 0 if no pair has been output. (When a pair is output on line 14, t is effectively updated.) The invariants are: 1) If $P_j \neq 0$, then j is not a final state and the string $x[P_j, i]$ is the smallest suffix of $x[t+1, i]$ which specifies a path in M from the start state to state j. 2) If $P_j = 0$, there is no suffix of x[t+1,i] which specifies a path in M from the start state to j.

Within the body of the loop at lines 3–20, the array P' is used to compute the updated value of P based on the previous value of P. Lines 18–19 effect the update.

The invariants for P' over the loop at lines 7–9 are as follows: 1) If $P'_k \neq 0$, then $x[P'_k, i]$ is the smallest suffix of $x[t+1, i]$ which specifies a path in M from the start state to k where the last transition is from a state numbered j or lower. 2) If $P'_k = 0$, then there is no path in M from the start state to k where the last transition is from a state numbered j or lower. Thus, after line 9, if $P'_j \neq 0$ then $x[P'_j, i]$ is the smallest suffix of $x[t+1, i]$ that specifies a path in M from the start state to j, and if $P'_j = 0$ then there is no suffix of $x[t+1, u]$ that specifies a path in M from the start state to j.

After line 9, there may be final state for which $P'_j \neq 0$. If this is the case, the loop on lines 10–12 discovers the largest u such that $x[u, i]$ is an element of L, thus $x[u, i]$ is an element of $G(L)$. The lines 13–17 output (u, i) (implicitly setting $t \leftarrow u$) and invalidate all partial or complete matches starting at or before u by setting the appropriate elements of P' to 0. This implies that after line 17, $P'_j = 0$ if j is a final state.

If (u, v) is a element of $G(L)$ it will the shortest suffix of x[t+1,i] for some t and will be output at line 14. □

5.4 Explicit Containment

A regular expression may be used to define an explicit universe for search. Using the search restriction advocated by this thesis the regular expression

<speaker> Σ^* </speech>

defines the universe of speeches. In his description of the `sam` text editor, Pike makes a distinction between the use of regular expressions for search and for extraction (retrieval) [82]. Using regular expressions we may specify both a universe of elements for retrieval and patterns for searching this universe.

We re-introduce the "containing" ($\rhd$) operator from the previous chapter to express a search over an explicit universe. The regular expression $r \rhd s$, where r and s are regular expressions, is defined as $G(r)\ \&\ (\Sigma^* s \Sigma^*)$.

Consider the query

Find speeches by witches that contain the words "Dunsinane" or "Birnam".

Using the containing operator and assuming our search restriction this query may be formulated using a regular expression as:

$$(\texttt{<speaker>}\ \Sigma^*\ \texttt{</speech>}) \rhd (\texttt{<speaker>}\ \Sigma^*\ \texttt{Witch}\ \texttt{</speaker>}\ \Sigma^*\ (\texttt{Birnam} \mid \texttt{Dunsinane}))$$

To this point we have not discussed the issue of converting a regular expression to finite automata, but some explanation is required in connection with explicit containment. In practice, the conversion is accomplished using a variant of a widely-used technique [5, 99]. Generally, the only differences being that the NFA is constructed with no λ-transitions and is constructed and maintained as a list of these transitions. Of some concern is the size of the NFA that will result from this conversion. The size of an NFA grows additively for alternation, concatenation and the varieties of repetition; for inclusion it grows multiplicatively. For most applications this growth is not a problem. Unfortunately, in order to implement exclusion, the NFA must be converted to a deterministic finite automata, with a possible exponential increase in size. It is then not reasonable to implement the "containing" operator by directly using the relationship in the proof of Theorem 5.1.

Fortunately, explicit containment may be implemented without direct use of that equation. We observe that the regular expression $r \rhd G(s)$ is equivalent to $r \rhd s$. It is thus possible to implement explicit containment by running two concurrent copies of the algorithm of figure 5.1, reporting a match to r only when it contains a match to s. This technique may also be used to implement the "not containing" ($\not\rhd$) operator.

In contrast, the "contained in" ($\lhd$) and "not contained in" ($\not\lhd$) operators cannot be implemented directly as regular expressions. Consider searching for $\texttt{ab} \lhd \texttt{a}\Sigma^*\texttt{c}$ in a text of the form

$(\mathtt{ab})^n\mathtt{c}$ for some fixed but arbitrary n using a single left-to-right scan of the text. Since the initial "a" might be (and in fact is) part of a match to $\mathtt{a}\Sigma^*\mathtt{c}$, each of the n matches to "ab" must be reported. These matches cannot be reported, however, until the final "c" is seen. Since n may be arbitrarily large, no constant store may be used to maintain the potential matches that would be discovered while this determination is being made. No single left-to-right scan with constant store can implement the $\triangleleft$ operator.

The final section of this chapter will describe the full adaptation of the structure algebra of Chapter 4 to pattern matching in un-indexed text. In that implementation, several simultaneous left-to-right scans are made, one for each term in the query expression, but only constant storage is required.

5.5 The `cgrep` Search Tool

The theory described in this chapter has been used as the basis for a search tool, `cgrep`, that has proved to be of significant value for several applications. In searching and organizing folders of mail messages and news articles, the tool allows the selection and extraction of articles based on combinations of the contents of the header lines and patterns occurring in the body. The tool has been used to assist in sorting, organizing and indexing structured documents from the TREC collection [54]. Other applications have included the extraction of World-Wide-Web hypertext links (URL's) from NetNews articles, searching files of machine code in binary format, searching comments in source code files, and counting actual occurrences of patterns in files (as opposed to lines containing the patterns). Formulating equivalent operations with existing text processing tools has proven difficult and frustrating.

The regular expression syntax of `cgrep` is based on that of POSIX 1003.2 extended regular expressions [59]. This regular expression syntax is based in turn on that of `egrep`. In addition, `cgrep` provides macros, an intersection operator, and several other useful extensions. In particular, the characters "<" and ">" may be used to match the beginning and end of file respectively. A detailed man page for the `cgrep` search tool is included as part of the software distribution [77]. In the remainder of this section the function and performance of the tool is explained by way of several examples.

5.5.1 Basic Searching

We begin with simple document processing examples.

Example 1: Multiple Matches within a Line

Sometimes interesting text patterns can be quite short and multiple instances may be found on a single line. The command

```
cgrep '\\cite\{.*\}' paper.tex
```

will pull a list of citations from a document in LaTeX format. If multiple citations appear on a line, all will be reported. Occurrences of "\", "{" and "}" appearing in the pattern are escaped with the backslash character "\" to prevent special meanings they have to `cgrep`. In all cases, an escape sequence consisting of a "\" followed by any punctuation character may be used to represent the literal punctuation mark, avoiding any special meaning of the character.

Example 2: Multi-Line Matching

Matching across lines is straightforward with `cgrep`. The command

```
cgrep '^.*United[[:space:]]+States.*$' constitution.txt
```

matches lines containing the name of the country. The two words composing the name may be separated by one or more white-space characters. The bracket expression "`[[:space:]]`" matches any character for which the *ctype* macro `isspace` would return true. This search will report both

```
We the people of the United States, in order to form a more
```

and

```
age of thirty years, and been nine years a citizen of the United
States and who shall not, when elected, be an inhabitant of that
```

The shortest match guarantees that only the smallest numbers of lines necessary are printed.

Example 3: Extracting Structural Elements

Under some mild assumptions, the command

```
cgrep '/\*.*\*/' *.c
```

will print all comments in a group of C source files. The character "`*`" is escaped, by preceding it with a backslash, to prevent its interpretation as a closure operator. The assumptions necessary for the extraction to work correctly are basically that comments contain no nested start patterns ("`/*`"), that the comment start and end patterns do not appear in string constants, and that comments are not ill-formed in conditionally-compiled code (e.g. in "`#ifdef`'s"). Many comments in C source code will satisfy these assumptions and it is possible to write `cgrep` expressions to detect exceptions — an example will be provided later.

Example 4: Searching SGML Text

SGML files usually contain substantial structure that is not line oriented. HTML documents provide an example of documents formatted (approximately) according to SGML conventions. The command

```
cgrep -i '\<title\>.*\</title\>' /users/*/public_html/*.html
```

extracts the titles of HTML documents from the top level of users' publicly accessible World-Wide-Web documents. The "`-i`" flag removes lower and upper case distinctions for matching purposes. The "`<`" and "`>`" characters are escaped, as they are used by `cgrep` to match file start and end characters. In this example and in previous examples the shortest substring matching guarantees that the patterns represented the beginning and end of a match are properly associated with one another.

Under longest-match disjoint substring search, the command of Example 2 would report a single match for any text containing the name of the country. The match would cover the entire file, starting at the beginning of the first line and continuing to the end of the last line. The command of Example 1 would report a single match starting at the beginning of the first citation and continuing to the end of the last closing brace. Similar problems occur with Examples 3 and 4.

By Theorem 5.1, it is always possible to re-write a regular expression explicitly to exclude longer matches. This is quite easy in some cases, such as the second example, where the pattern

```
'^[^\n]*United[[:space:]]+States[^\n]*$'
```

is sufficient, assuming that the escape sequence "`\n`" matches the newline character.

Example 3 requires more work. It's necessary to recognize the end pattern "`*/`" while allowing the comment to include "`*`" and "`/`" characters that are not part of the end pattern. One possibility is:

```
/\*([^/*]|/+[^/*]|\*+[^/*])*(/*|\**)\*/
```

The other examples exhibit similar difficulties.

Example 5: Intersection of Patterns

In addition to the standard POSIX 1003.2 operators, `cgrep` accepts "`&`" for the intersection of two regular expressions. The precedence of the intersection operator is the same as that of union ("`|`"). The union and intersection operators associate left to right.

The intersection operator "`&`" provides a way to match intervals of text that satisfy a combination of patterns. The command

```
cgrep '.{0,100}&.*God.*&.*Moses.*' exodus.txt
```

prints all intervals of text that are 100 characters or less in size and contain both "God" and "Moses". Allowing overlap between matches is important so that passages such as

```
Moses took the
        rod of God
```

and

```
God in his hand.
```

```
004:021 And the LORD said unto Moses
```

can both be reported.

5.5.2 Search Universes

The "`-U`" option of `cgrep` is used to specify a search universe. A second regular expression is used to select members of the search universe that contain an instance of the regular expression. Through this mechanism, `cgrep` provides a single $\triangleright$ operator applied at the top level.

Example 6: Lines

With an appropriate universe definition, `cgrep` can mimic the behavior of standard Unix search utilities. The command

```
cgrep -U '^.*$' '[Vv]egetarians|[Vv]egans' mbox
```

prints all lines that contain references to vegetarians or vegans. Using the search universe of lines, the capabilities of `cgrep` are a superset of those of `egrep`.

Example 7: Groups of Lines

Groups of several lines can be used as a search universe. In this circumstance, the utility of allowing overlapping matches becomes obvious, as each group of lines will overlap with several others. The command,

```
cgrep -U '^.*^.*^.*$' '.*God.*&.*Moses.*' exodus.txt
```

will print groups of three lines that contain both "God" and "Moses".

At this point, it is worth discussing the exact meaning of the "`^`" and "`$`" operators in `cgrep` and the default behavior for printing matches. In standard search tools, the "`^`" operator matches the beginning of a line and "`$`" matches the end of a line. Since `cgrep` allows matching across line boundaries, "`^`" and "`$`" have been adapted slightly: "`^`" matches a newline character or the beginning of the file; "`$`" matches a newline or the end of the file. In other words, "`^`" is equivalent to "`\n|<`" and "`$`" is equivalent to "`\n|>`".

Following these definitions, each match reported by the command of Example 6 would be preceded and terminated by a newline character. Visually, each match would appear to be preceded by a blank line.

We have found it convenient to automatically strip off these leading newlines. Similarly, it has been found convenient to terminate matches that would not normally end in a newline by adding a newline character. This automatic reformatting of the output is particularly helpful when the results of a `cgrep` search are to be piped to other Unix commands.

An option ("`-binary`") is provided to prevent automatic reformatting of the output. The "`-binary`" option also changes the meaning of the "`^`" and "`$`" operators so that they match only the beginning and end of file, becoming equivalent to the "`<`" and "`>`" operators. In addition to automatic reformatting, an option ("`-tag`") allows the user to specify start and end tags to be placed around matches.

Example 8: Mail Messages

The command

```
cgrep -U '^From .*(^From |>)' 'vegetarian' mbox
```

extracts all mail messages that contain the word "vegetarian". Mail messages are assumed to start with the word "From" at the beginning of a line followed immediately by a space character. A mail message ends when another of these patterns is seen or the end of file is encountered. Unless it occurred at the end of the input file each match reported will be terminated by a line containing the word "From" followed by a space. In order to create a validly formatted mail folder the output must be piped through

```
sed '/^From $/d'
```

or equivalently

```
cgrep -V '^.*$' '^From $'
```

The latter command prints all lines that do not contain the pattern "`^From $`". The `-V` option is used to specify an *anti-search universe*, which is the subject of the next example.

Example 9: Anti-Search Universes

Members of an anti-search universe are reported if they do not contain a match to a second regular expression. The command

```
cgrep -list -V '/\*.*/\*' '\*/' *.c
```

will print the names of files containing a comment with a nested comment-start pattern ("`/*`"), violating one of the assumptions of Example 3. The "`-list`" option specifies that only the names of files containing matches be reported. Through this mechanism, `cgrep` provides a single $\not\triangleright$ operator applied at the top level.

5.5.3 Other Features of `cgrep`

Several other features of `cgrep` assist in its use: A macro facility is provided to help formulate search and retrieval operations involving commonly-used subexpressions. As already mentioned, a tagging facility allows the user to define delimiters to be placed around each match reported.

The syntax for macros is slightly ugly. Macro calls come in two varieties: *fast* and *tedious*. A fast call consists of a "`@`" character followed by an single alphabetic character. A tedious macro call has the form:

```
[@name(parameter0, parameter1, ...)]
```

where each of the up to 9 parameters is a regular expression. If the macro requires no parameters, the bracket-enclosed parameter list is omitted completely.

Macro definitions are placed in files that are read on program start-up. Any definitions in the file `".cgreprc"` in the user's home directory are read first. The "`-defs`" option may be used to specify additional macro files. The definition of fast and tedious macros are identical. Any un-parameterized, single-letter macro is automatically usable as either a fast macro or a tedious macro.
An un-parameterized macro definition has the form:

```
name=regular-expression
```

and a parameterized macro definition has the form:

```
name#n=regular-expression
```

where the number of parameters is indicated by a single digit following the "#" character. Within the body of a parameterized macro, the actual parameters may be referenced as "`#1`" through "`#9`" A macro name must start with an alphabetic character, and may include only alphanumeric characters and the character "_".

Example 10: Macros

Standard macros for handling mail may be defined in `".cgreprc"` in the user's home directory:

```
Mail=^From .*(^From |>)
From#1=^From:[^$]*#1
Re#1=^Subject:[^$]*#1
```

The command

```
cgrep -U '[@Mail]' '(.*[@From([Cc]owan)].*)&(.*[@Re((fwd))].*)' mbox
```

would then extract all mail messages forwarded by Cowan.

Example 11: Tagging

Since the output from `cgrep` often consists of multi-line text fragments, the tagging facility allows matches to be easily distinguishable from one another and simplifies further searching on the output. The command,

```
cgrep -tag '<q>\n' '</q>\n'\
   -U '^.*^.*^.*$' '.*God.*&.*Moses.*' exodus.txt
```

will tag each excerpt in the style of SGML. An "@" appearing in a tag is replaced by the name of the file where the occurrence was found. Any escape sequence valid in a regular expression (such as "`\n`") is valid in a tag. Thus, in this example, each of the tags generated by the command will appear alone on a line.

Example 12: Searching Binary Files

Files of binary data may be searched using `cgrep`. The command

```
cgrep '[^[:print:]][[:print:]]{4,}[\n\0]' cgrep
```

reports strings of four or more printable characters ending in a newline or null character that appear in the executable file `"cgrep"`. This search is similar that performed by the Unix `strings` command.

Escape sequences allow non-printable characters, such as the null character ("`\0`"), to be used in a pattern. These escape sequences follow the syntax of ANSI C, including the sequences for hexadecimal and octal constants. Escape sequences not given special meaning by ANSI C always represent the literal character following the "\".

Each result reported by `cgrep` will begin with a non-printable character and end with a newline or null character. This example further illustrates how a longest-match requirement ("four or more") can be satisfied with a shortest-match search.

5.5.4 Performance

The table in figure 5.2 gives performance figures for the examples running over typical input text, reading and writing to a disk file. The programs were executed on a SPARCServer 670MP under SunOS 5.3 concurrently with the "normal" activities of a dozen or so other users, mostly Programming Language and Theory of Computation researchers in the Computer Science Department at the University of Waterloo. The input sources described as "US Constitution" and "Exodus" are electronic texts from Project Gutenburg [83]. The LaTeX source is that of reference [27]. The source to `nethack` supplied an example of C source code with lots of oddly structured comments. The HTML source is a collection of local users' World-Wide-Web home pages. The personal email was received by the author over several months. Elapsed times include both system and user times, and are averaged over several runs.

The figures show that the program is generally quite usable to search large files interactively. One exception is the command of Example 5, which processed its input at less than 1KB per second. This poor performance is due to a large number of active states during the matching process. The same command searching the entire Old Testament would take more than an hour. One solution to this problem is to pre-filter the input with a command such as that of Example 11, tagging each group of lines, and then running the actual search on the result.
The performance of `cgrep` is often inferior to more specialized tools on equivalent tasks. The command

```
egrep  '[Vv]egetarians|[Vv]egans' mbox
```

Example Number	Input Description	Input Size (bytes)	Number of Matches	Output Size (bytes)	Elapsed Time (seconds)
1	LaTeX source	68K	33	0.49K	0.13
2	US Constitution	46K	85	5.2K	0.23
3	`nethack` source	1.6M	4647	231K	2.9
4	HTML source	544K	79	3.5K	0.55
5	Exodus	193K	49	2.6K	220
6	Personal email	12M	331	21K	11
7	Exodus	193K	88	12K	3.1
8	Personal email	12M	214	2.8M	46
9	`nethack` source	1.6M	90	0.75K	0.47
10	Personal email	12M	17	164K	138
11	Exodus	193K	88	13K	3.1
12	`cgrep` executable	46K	430	7.5K	0.27

Figure 5.2: Performance of `cgrep`.

is equivalent to the `cgrep` command of Example 6 but is more than twice as fast, running in just over five seconds on the same input. The Unix command

```
strings -a cgrep
```

is equivalent to the `cgrep` command of Example 12 but is nearly three times faster, running in just under a tenth of a second. On the other hand, `egrep` and `strings` cannot perform the search and retrieval operations in the other examples.

In other cases, `cgrep` has proven to be faster than special purpose programs. The program `readmsg` is part of the *elm* mail package and can be used to extract mail messages based on patterns in the message body. The command

```
readmsg -ah -f mbox vegetarian
```

performs the same function as the `cgrep` command in Example 8 but takes 56 seconds. In this case, `cgrep` is 17% faster. Furthermore, `readmsg` cannot be used to perform more complex retrieval operations, such as that of Example 10.

5.6 The Structure Algebra Revisited

In this final section of the chapter we describe an implementation of the structure algebra of the previous chapter, assuming left-to-right scanning of un-indexed text. This implementation of the

algebra may also be used with indexed text by scanning the postings list for each term, rather than indexing into them. Under this approach, the time to generate all solutions to a query is $O(\Sigma m_i)$, where m_i is the length of the postings list for the ith term.

An implementation of the algebra for flat text search provides several benefits. As discussed, the "contained in" ($\lhd$) and "not contained in" ($\not\lhd$) operators cannot be implemented in terms of regular expression search, and direct implementation of the "containing" ($\rhd$) and "not containing" ($\not\rhd$) operators requires the possibly-expensive construction of a DFA from a NFA. Further, a direct use of the "both of" ($\triangle$) operator may improve the search time of the query in example 5, which then could be written

$$(\texttt{'God'} \triangle \texttt{'Moses'}) \lhd \texttt{'.\{100\}'}$$

In the context of the MultiText project, an implementation of the algebra over flat text is used to assist in the process of marking-up query results with HTML for display by a World-Wide-Web browser.

To implement the structure algebra, we maintain for each node Q in the expression tree for a query the four attributes

$$Q.\mathrm{p},\ \ Q.\mathrm{q},\ \ Q.\texttt{pnext},\ \ Q.\texttt{qnext}$$

where the pair

$$(Q.\mathrm{p}, Q.\mathrm{q})$$

represents the "current" solution to Q and the pair

$$(Q.\texttt{pnext}, Q.\texttt{qnext})$$

represents the "next" solution to Q.

Query terms are regular expressions. The implementation of τ for regular expressions is based on figure 5.1. We assume that each term in the query is running a version of this algorithm with its data structures specialized to the particular regular expression being evaluated and re-organized so that solutions are returned one-at-a-time rather than being directly output, with (∞, ∞) being returned when no solutions remain. If this function is called `search`, an implementation of the access function τ for a regular expression r is shown in figure 5.3.

```
τ (r, k) =
 begin
   while Q.p < k do begin
     (Q.p, Q.q) ← (Q.pnext, Q.qnext);
     (Q.pnext, Q.qnext) ← search(r);
   end;
   Return (Q.p, Q.q);
 end;
```

Figure 5.3: τ for regular expression pattern matching.

The implementation of τ for the operators of the structure algebra is taken from the equations of figure 4.4, figure 4.5 and figure 4.6. With two exceptions, the equations for ρ, τ' and ρ' are not used. The equation for $\rho\,(A \rhd B, k)$ is used in the body of the equation for $\tau\,(A \rhd B, k)$; the equation for $\rho\,(A \not\rhd B, k)$ is used in the body of the equation for $\tau\,(A \not\rhd B, k)$. In both cases, we treat the equation for ρ as being defined within the **let** statement of the equation for τ, restricting the scope of the definition. None of the equations for ρ, τ' or ρ' are used directly. Instead, we define ρ and τ' in terms of τ, as shown in figure 5.4 and figure 5.5. No equivalent definition for ρ' is required, since ρ' is not called directly by the equations for τ. The driver procedure of figure 4.14 may then be used to generate the solutions.

For each term in a query, the implementation makes a left-to-right scan in tandem over the text. Storage requirements are proportional to the size of the query, but do not depend on the size of the text.

```
ρ (Q, k) =
 begin
   while Q.q < k do
     τ (Q, Q.p + ε);
   Return (Q.p, Q.q);
 end;
```

Figure 5.4: ρ for structural pattern matching.

```
τ′ (Q, k) =
 begin
   while Q.qnext ≤ k do
     τ (Q, Q.p + ε);
   Return (Q.p, Q.q);
 end;
```

Figure 5.5: τ' for structural pattern matching.

Chapter 6

Shortest Substring Ranking

Section 2.1 discussed two query models for document database systems. The Boolean query model allows a user to precisely specify a document set for retrieval using standard set operations. In contrast, the statistical query model orders all the documents in the collection according to their statistical similarity with the user's query. Ideally, this ordering corresponds to users' opinions of the relative relevance of documents to their queries, and approaches to statistical ranking may be evaluated on this basis.

In this chapter, the principal property of the shortest substring search model is exploited to develop a combined approach for searching document databases. Boolean queries are formulated using the combination operators of the structure algebra, with the "both of" operator ($\triangle$) corresponding to the Boolean AND operator and the "one of" operator ($\bigtriangledown$) corresponding to the Boolean OR operator. A document contains a solution to such a query if and only if it is a member of the document set that would be selected by the equivalent Boolean query. Documents containing solutions are ranked according to the lengths of the solution extents contained within them.

6.1 Ranking of Boolean Queries

The lack of relevance ranking is widely seen a significant limitation of the Boolean query model [41, 84, 94, 103]. The case for relevance ranking becomes particularly strong when a Boolean query matches many documents, forcing the user to rephrase the query or scan through a large amount of undifferentiated material.

One simple approach is to select a document set using a Boolean query and then rank these documents using a statistical technique based on the query terms alone, ignoring the Boolean operators for ranking purposes [79]. Unfortunately, since the Boolean operators are ignored, term relationships described by the operators are not reflected in the ranking, and the results can be counterintuitive [84].

Waller and Kraft [103] follow an approach based on fuzzy set theory and discuss how the Boolean query model may be extended to take advantage of term weights calculated from document characteristics and on weights explicitly added to query terms by the user. The *p-norm* model [94] provides a spectrum of ranking techniques, in which the relationships described by the Boolean operators can be of varying importance. At one extreme, Boolean relationships are ignored and the approach is equivalent to the inner product measure. At the other extreme, the approach is equivalent to the fuzzy set model, with documents matching the Boolean expression exactly. Intermediate between these extremes, the approach has been shown to produce results that can be superior to either the inner product measure or the fuzzy set model.

6.2 Ranking by Solution Density

Under the shortest substring model, the result of a query is the set of smallest extents that satisfy a specified predicate. The size of this interval may be used as a measure of the "goodness" of the match, particularly in the case of the Boolean-flavored queries that are expressible with the combination operators. For example, if our query is

> *Find fragments of text that contain both "Nixon" and "Watergate".*

it is not unreasonable to expect that the proximity of these terms is positively correlated with our interest in the text. If we consider a containing document, the most relevant might be expected to contain many elements from the solution set and for these elements each to be relatively short.

Ranking is based on two assumptions:

- *Assumption A*
  The smaller a solution extent, the more likely that the corresponding text is relevant.
- *Assumption B*
  The more solution extents contained in a document, the more likely that the document is relevant.

The first assumption provides a ranking for individual solution extents; the second suggests a ranking technique for particular documents in terms of solution extents. Both assumptions are superficially reasonable. However, a problem remains in combining these assumptions to produce a single score for a document.

Assumption B suggests that a document might be ranked by summing individual scores of solution extents contained within it. A natural value to use as the score of a particular extent (p, q) is its length $|(p, q)| = (q - p + \epsilon)$. Unfortunately this approach assigns a higher score to less relevant document. Summing individual scores is reasonable only if a higher score indicates a more relevant document. The problem is rectified if an inverse relationship is used.

$$\text{Score of } (p, q) \propto \frac{1}{|(p, q)|}$$

or

$$\text{Score of } (p, q) = S(p, q) = \frac{\mathcal{K}}{|(p, q)|}$$

During preliminary trials of the technique it was quickly observed that if the length of an extent was below a threshold of a dozen or so words, assumption A no longer appeared to hold and all extents appeared equally relevant — certainly not varying in relevance at the level indicated by an inverse relationship.

Therefore, we take the score for a particular extent as:

$$S(p, q) = \begin{cases} \frac{\mathcal{K}}{|(p,q)|} & \text{if } |(p, q)| \geq \mathcal{K} \\ 1 & \text{if } |(p, q)| \leq \mathcal{K} \end{cases}$$

For any extent (p, q), we have $0 < S(p, q) \leq 1$. If solution extents (p_1, q_1)...,(p_N, q_N) are contained in a particular document, the score for the document is

$$\sum_{i=0}^{N} S(p_i, q_i)$$

6.3 TREC-4 Experiment

This section summarizes MultiText participation in the fourth Text REtrieval Conference (TREC-4). The focus of the TREC conferences is the experimental evaluation of new Information Retrieval technologies. MultiText used TREC-4 as a forum for the introduction and evaluation of

shortest substring ranking. This summary includes a brief overview of TREC-4. For a complete description of TREC-4 experiments, and the activities of other participating groups, refer to the full conference proceedings [52].

6.3.1 Overview of TREC

Since 1992, the (U. S.) National Institute of Standards and Technology (NIST) has sponsored the TREC conferences to provide for common comparison and evaluation of Information Retrieval systems. Participating groups come from both industry and academia. The conference is organized around experimental participation in one or more document ranking tasks.

Each task is structured similarly. Participants are given a set of user interest statements, called "topics", and a document collection. For example, TREC topic 246 is shown in figure 6.1. Participants formulate queries from these topics, either manually or through automatic processing, and submits these queries to their systems for evaluation over the document collection. For each topic, participants report a scored list of the 1000 documents judged to be the most relevant by their system. Submissions from all participants are then subjected to a common evaluation procedure.

Evaluation is made on the basis of two standard *effectiveness* measures: *precision* and *recall*. Given a set of documents $\mathcal{D}$, drawn from a collection $\mathcal{C}$, these two measures are defined as follows:

$$\textit{precision} = \frac{\textit{number of relevant documents in } \mathcal{D}}{\textit{total number of documents in } \mathcal{D}}$$

$$\textit{recall} = \frac{\textit{number of relevant documents in } \mathcal{D}}{\textit{number of relevant documents in } \mathcal{C}}$$

Given a document ranking, precision and recall may be measured for the top r documents. Values of precision and recall change as r is varied. Recall will remain constant or increase as r is increased, with precision typically falling. By comparing precision at various recall levels, approaches to ranking may be compared.

For the TREC experiments, the number of relevant documents in the collection is estimated for each run by pooling the top 100 documents submitted by each group. All relevant documents are assumed to be contained in this pool. Independent assessors judge the relevance of the documents in the pool, and these relevance judgements form the basis of the recall and precision measurements.

The TREC experiments are organized into two main tasks, the "ad-hoc" task and the "routing" task, and several special-interest tasks. The ad-hoc task simulates a user's initial interaction with an Information Retrieval system. Queries are developed from the topics without reference to the document collection and without additional relevance information. For example, if queries are developed manually, the searcher may not interactively refine the query by using feedback from initial runs to add query terms or to re-structure the query. The routing task simulates an on-going user interest. Routing topics are drawn from previous TREC conferences, and participants may use previous relevance judgements and interaction techniques to develop and refine their queries. Once queries are finalized, participants are given an entirely new document collection on which the queries must be run without further refinement. Beyond the main tasks, special purpose tracks address special-interest issues including user interaction, multi-lingual collections and data corruption.

6.3.2 Procedure

The MultiText Project participated in both the routing task and the ad-hoc task. Queries were developed manually; the procedure differed only slightly for the two tasks. Approximately 15 to 45 minutes were spent developing a query for each topic. During creation of the routing queries, relevant documents were occasionally pulled and used as a source of possible terms, but this practice was not uniformly followed. Besides personal knowledge, the only external resources used were an on-line dictionary (Webster's); the Unix `spell` program; an on-line list of country, state and city names and state postal abbreviations; and, in a few cases, current issues of newspapers.

The final query developed for each topic was a compound query consisting of an ordered list of one or more sub-queries (2.05 sub-queries on average). Results for each sub-query were determined separately using the ranking techniques described in the previous section. For the TREC experiments, a value of 16 was used for the constant $\mathcal{K}$. The results were then combined into a final solution set according to the ordering of the sub-query list, with results of a particular sub-query ranked before the results of subsequent sub-queries. Documents given a non-zero score by one sub-query were eliminated from the results of subsequent sub-queries before this final ranking.

This approach reflects a trade-off between a desire for precision and an artificial need to produce 1000 ranked documents. The query appearing at the beginning of the list is intended to be a precise expression of the requirements underlying the topic. Queries occurring later in the

list are "weaker" and are intended to pick up a large number of possibly relevant documents.

Figure 6.1 shows topic 246 and figure 6.2 gives the corresponding query in the internal format of the MultiText system, which was forwarded to NIST. Some explanation of the syntax is required: "`^`" and "`+`" are equivalent to "both of" ($\triangle$) and "one of" ($\triangledown$) operators respectively, "`<>`" is equivalent to the ordering operator ($\diamond$), the expression "`[2]`" represents fixed-length intervals two words in length (Σ^2), and "`<`" represents the "contained in" operator ($\lhd$). The "`@output`" command sets the output file name for the query. The "`@rank`" command takes a topic number and a compound query as arguments and executes the ranking procedure. The topic number is used by the "`@rank`" command only for formatting the output. In this case the compound query list has only a single sub-query, which has been assigned the name "`q`".

The query of figure 6.2 is essentially a Boolean expression in conjunctive normal form, consisting of three "facets" [33], each built from several named pieces for convenience. The first facet ("`arms`") is a disjunction of terms and phrases related to military weapons. The second facet ("`export`") is a disjunction of terms related to trade. The final facet ("`USbroad`") is a disjunction of 150 geographical place names and abbreviations related to the United States. The definition of this last facet is not included in figure 6.2; its definition is global in scope, and it is used whenever a topic concerns only the U.S.

Several other global definitions of this type were used in developing the queries, and these definitions contributed significantly to the size of the queries. For the ad-hoc task, queries contained an average of 67 terms. For the routing task, queries contained an average of 53 terms. The variance was fairly high. For some topics the query consisted of hundreds of terms; for other topics the query consisted of a single two-term phrase. Overall, about half of the query terms resulted from the expansion of global definitions, overwhelmingly from the expansion of the "`USbroad`" definition. Expansion of phrases into terms and the inclusion of term variants also contributed to the large number of terms per query.

Query evaluation proceeds in five steps:

1. Determine extents which satisfy the specified query.

2. For each solution extent, either determine the document extent that contains it, or if it overlaps document boundaries eliminate it.

   $$\text{DOCUMENTS} \rhd \text{Q}$$

3. For each document containing solution extents, determine the identifier for the document.

   $$\text{ID} \lhd (\text{DOCUMENTS} \rhd \text{Q})$$

```
<top>

<num> Number:  246

<desc> Description:

What is the extent of U.S. arms exports?

</top>
```

Figure 6.1: TREC topic 246.

```
@output "246.output"

arms0 = "arms" + "gun" + "guns" + "tanks"
arms1 = "firearm" + "firearms" + "weapon" + "weapons" + "rifle" + "rifles"
arms2 = (("fighter" <> ("jet" + "jets")) <  [2]) + "bomber" + "bombers"
arms = arms0 + arms1 + arms2

export0="export" + "exports" + "trade" + "sale" + "sales"
export1="tariff" + "tariffs"
export=export0 + export1

q = armsˆexportˆUSbroad

@rank 246 q
```

Figure 6.2: Query for topic 246.

4. For each document containing solution extents, calculate a document score using the scoring method of Section 6.2.

5. Sort the document identifiers by document score.

The MultiText experiments were run on a dedicated DEC Alpha 2000/300 running OSF V3.2 with 64MB of RAM.

6.3.3 Results and Discussion

Results for the ad-hoc task were quite good. Average precision was third-best among the participating groups. For over 65% of the topics the average precision was above the median average precision for all groups. Unfortunately, no valid results can be reported for the routing task, as approximately one-third of the required test data was not included in the database when the submitted run was generated.

Figure 6.3 compares the ad-hoc runs submitted by each of the participating groups. Groups were permitted to submit two runs, and in cases where two runs were submitted, the better of the two is plotted in the figure. The plots are generated directly from data supplied by NIST, with precision plotted against recall. Exact precision values are interpolated to 10% recall increments by taking the maximum precision achieved at all recall levels greater than or equal to the level of the increment. These interpolated precision values always decrease with increasing recall. When comparing the results, it should be borne in mind that the represented systems used extremely different techniques, some of which are well established and some of which are quite experimental. Because of the diversity of the systems and the limited nature of the experiment, no general conclusions about the relative quality of the systems should be drawn. However, it does appear that shortest substring ranking can be competitive with the best known practice.

The ranking technique is reasonably efficient. For the ad-hoc task, system search time required an average of 40 seconds elapsed time per query (an average of 18 seconds per sub-query). For the routing task, system search time required an average of 10 seconds elapsed time per query (an average of 5 seconds per sub-query). After the results were submitted to NIST, simple performance tuning, requiring less than an evening's work, made the runs 38% faster (with an average of 29 seconds elapsed time per query, 13 seconds per sub-query, for the ad-hoc task). This improvement was reported in the MultiText TREC-4 paper [29]. Since that time, the MultiText system has undergone additional enhancements that have reduced search time to an average of 22 seconds elapsed time per query (an average of 10 seconds per sub-query) for the ad-hoc task.

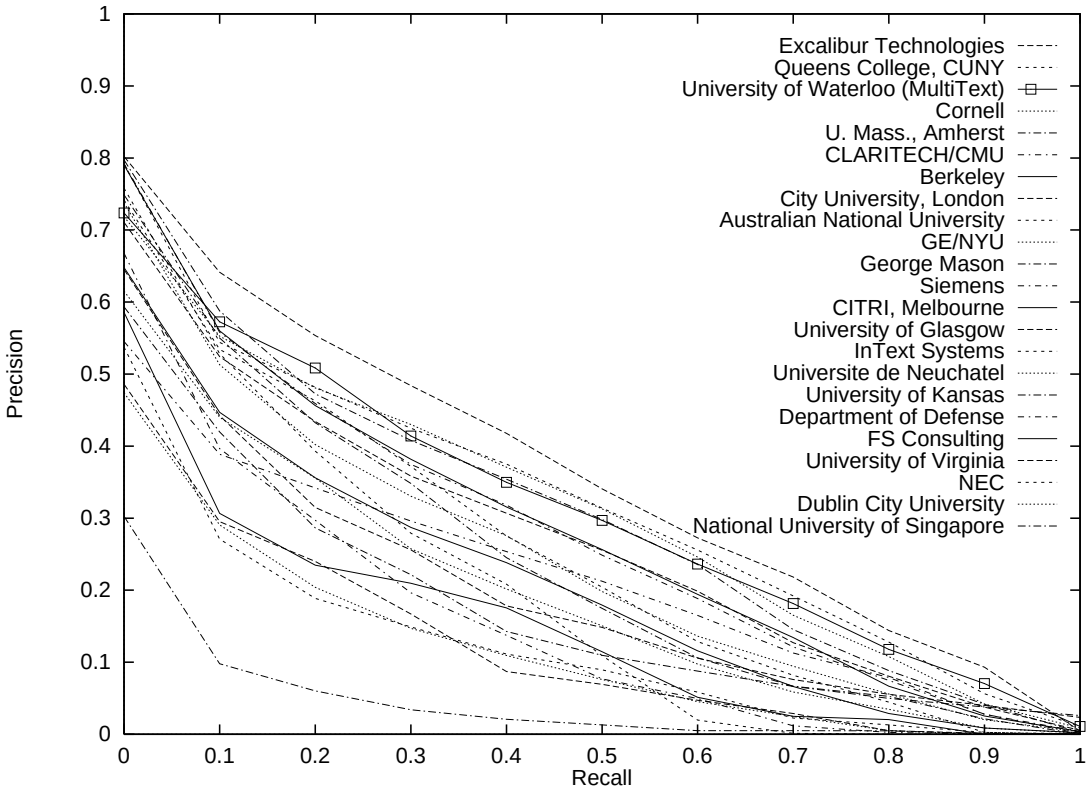


Figure 6.3: Comparison of TREC-4 ad-hoc results.

Furthermore, shortest substring ranking produces scores that are independent of the characteristics of other documents in the collection, a property not generally shared by statistical ranking techniques [102]. This property allows a collection to be split arbitrarily into a number of subcollections to be searched in parallel by separate search engines, with the results merged for the final ranking. This approach would produce essentially a linear speed-up in search time with the number of engines used.

Figure 6.4 shows equivalent timings for other systems. These timings were not formally measured as part of the TREC experiment, and were reported for information purposes as part of system descriptions. The systems used a wide variety of hardware, including highly-parallel machines and supercomputers, and the timings are not normalized with respect to hardware. Moreover, because of ambiguity in the information questionnaire some groups may have reported only CPU time, and not included I/O latency time. Due to these confounds, timing comparisons must be made with extreme care. However, it does appear that shortest substring ranking can be competitive in efficiency with the fastest systems.

6.4 Further Related Work

As part of TREC-3 [51], Charoenkitkarn, Chignell and Golovchinsky [23] produced reasonable preformance using a simple boolean ranking technique along with highly interactive query development. Also as part of TREC-3, Hawking and Thistlewaite [55] described experiments using the PADRE parallel free-text scanning system. For ranking, they used a weighted sum based on term frequency and document length.

As part of TREC-4, Hawking and Thistlewaite [56] continued their work with the PADRE system using a proximity scoring technique similar to shortest substring ranking, but with fixed proximity limits defined as part of the query. Solutions longer than this fixed limit were not considered during the ranking process. Further, they appear to have considered nested solutions as well as overlapping solutions. Their scoring technique produced good precision-recall results. In Figure 6.3 their results are labeled as “Australian National University”.

System	Search time
Excalibur Technologies	N/A
Queens College, CUNY	N/A (3–4 minutes for routing runs)
University of Waterloo (MultiText)	40 seconds
Cornell	N/A (TREC-3 ad-hoc: 1–4 minutes)
U. Mass., Amherst	N/A (TREC-3 ad-hoc: $\approx$ 4 minutes)
CLARITECH/CMU	$\approx$ 6 hours (parallel hardware, no indexing)
Berkeley	$\approx$ 30 seconds
City University, London	$<$ 60 seconds
Australian National University	N/A (parallel hardware; no indexing)
GE/NYU	60 seconds
George Mason	$\approx$ 60 seconds
Siemens	74 seconds
CITRI, Melbourne	$<$ 1 second (extremely short queries)
University of Glasgow	$<$ 1 minute
InText Systems	120 seconds
Université de NeuChâtel	15.5 seconds
University of Kansas	144 seconds
Department of Defense	$\approx$ 30 seconds
FS Consulting	18 seconds
University of Virginia	150 seconds
NEC	1200 seconds
Dublin City University	10 seconds
National University of Singapore	32.4 seconds

Figure 6.4: TREC-4 system performance.

Chapter 7

Conclusion

7.1 Closure

Operators in the structure algebra take GC-lists as operands and produce GC-lists as results. Closure of the algebra over GC-lists unifies the contributions of the thesis. Previous systems for specifying search over structured text required solution elements to be drawn from pre-defined sets specified by a meta-structural description or had significant semantic problems directly related to closure (Section 2.3).

The specific contributions of the thesis are as follows:

- A new model for representing structure in text (Section 2.2.5). Under the model, structure is recorded without any requirement for a formal description of the relationship between structural elements. The approach allows the incremental addition of structural descriptions, and provides a uniform representation for both logical and physical structure.
- The introduction of GC-lists as a new abstract data type for representing the results of a text search (Chapter 3). In general, a search over a text string may be satisfied by a quadratic number of substrings. GC-lists provide a method of linearizing sizes of solution sets by disallowing solution substrings that would have other solution substrings nested within them (Theorem 3.2). Other linearizing methods can introduce a left-to-right (or right-to-left) bias in the solution set (Section 5.1.2) and may require an examination of the entire text to determine if a particular substring is part of the solution set (Section 5.3.1). In addition, the elements of a GC-lists are totally ordered by their end points (Theorem 3.1), which permits direct indexing into a GC-list (Section 4.4).

- An algebra for structured text search (Chapter 4, Section 4.1).
- Proofs for various properties of the algebra (Section 4.3), including commutative, associative, and distributive laws, and two generalizations of its operators (Section 4.7).
- An implementation framework for the algebra (Section 4.4, Section 4.5). Under the statistical assumptions discussed in Section 4.6.1, the framework may be used to find individual solutions in $O(\log(m))$ expected time by indexing into GC-lists (Section 4.6). As an alternative, a modified version of the framework may be used to merge lists of index terms, with the time for finding all solutions linear in the sum of the length of the lists (Section 5.6).
- Application of the shortest substring search model to regular expression pattern matching (Chapter 5), providing a new method for choosing which matching substrings to report. Scanning a text and reporting the locations of matches requires only a single left-to-right scan with constant store (Section 5.3.2). This property is not shared by other approaches, particularly those that use the standard "left-most longest match" rule (Section 5.3.1). The practical value of a search tool based on the approach is demonstrated (Section 5.5).
- Application of the shortest substring property to relevance ranking (Chapter 6). Documents are ranked according to the length and number of solution extents contained within them. The ranking technique compared well with established approaches in an external experiment conducted by the National Institute of Standards and Technology (Section 6.3).

All of these results have been tested in practice in the context of the MultiText Project.

In addition to the contributions made directly by the thesis, the algebra makes it possible to give a precise formal semantics for the structural search operators of the PAT commercial text retrieval language (Figure 4.19). Prior published descriptions of PAT have been informal in nature [39, 43, 90] or have described a heavily modified version of the language [32].

7.2 Additional Properties of Generalized Concordance Lists

Several further properties related to GC-lists may be demonstrated. These results have no direct bearing on the main results of the present thesis, but may be of interest for future work.

By Theorem 5.1 regular languages are closed under G. The equivalent is not true for context free languages, and in particular for deterministic context free languages. Consider the determin-

istic context free language

$$K = \texttt{\^{}a}^n\texttt{b}^*\texttt{c}^n\texttt{\$} \cup \texttt{ab}^i\texttt{c}^j\texttt{\$} \cup \texttt{aab}^j\texttt{c}^i\texttt{\$} \cup \texttt{ac}$$

where n, i and j are natural numbers with $i > j$.

$$G(K) = \texttt{\^{}a}^n\texttt{b}^n\texttt{c}^n\texttt{\$} \cup \texttt{ab}^i\texttt{c}^j\texttt{\$} \cup \texttt{aab}^j\texttt{c}^i\texttt{\$} \cup \texttt{ac}$$

A simple pumping lemma argument [57, pages 125–128] may be used to show that $G(K)$ is not context free.

For a language L define

$$H(L) = L - G(L)$$

Further define

$$H^1(L) = H(L)$$

and

$$H^i(L) = H^{i-1}(H(L)) \qquad \forall i > 1$$

Let the *G-height* of a language L be the smallest n such that $H^n(L) = \emptyset$ if such an n exists; otherwise define the G-height of the language as ∞. The *GC-height* of a set of extents may be defined equivalently.

There exists an unlimited number of non-isomorphic regular languages of any given G-height. Let

$$s_1 = \texttt{a}r\texttt{b}$$

where r is any regular expression not involving "a" or "b", and let

$$s_i = s_{i-1} + (\texttt{a}r)^i\texttt{b}$$

The language defined by the regular expression s_n has G-height n. The language defined by the regular expression $(\texttt{a}r)^*\texttt{b}$ has G-height ∞. If r represents only the empty string, s_n represents a finite language with G-height n.

7.3 Future Work

We conclude with precis for research projects to extend the work of the thesis.

7.3.1 Query Plans and Solution Caching

The indexing strategy of Section 4.4 and the merging strategy of Section 5.6 represent markedly different approaches to solving a query in the structure algebra. Developing an optimization process for the structure algebra might involve selecting from these alternatives, perhaps choosing different strategies for different sub-queries.

The algebraic properties developed in Section 4.3 may be useful for re-writing queries to improve performance. For example, the distributive properties may be used to eliminate duplicated terms, and the relationships of Theorem 4.11 and Theorem 4.12 may potentially reduce the total number of calls to access functions.

GC-lists for frequently posed queries might be cached in memory or on disk. Queries that are candidates for caching might be recognized dynamically by examining the stream of users' queries, or might be specified statically by the database administrator. Queries for specific structural elements (documents, paragraphs, verses or lines) are particularly good candidates for caching (Section 4.4.1).

7.3.2 Recursive Nesting

The lack of support for recursive nesting (sections within sections within sections ...) has been a major criticism of the structure algebra and the shortest substring search model [35, 78, 98]. The example on page 31 shows one possible way of dealing with recursively nested structure. This example, together with the discussion of GC-height in the previous section, suggest the representation of a recursively nested set of extents as an ordered list of GC-lists, where each GC-list contains the extents of a particular GC-height and the GC-lists are ordered according to GC-height. In the example, the sub-query BLOCK0 represents blocks with GC-height 0, the sub-query BLOCK1 represents blocks with GC-height 1, and the solution to the overall query for triply-nested blocks has GC-height 2.

Dao et al. [35] have extended the containment operators of the structure algebra to general sets of extents. They impose a total order on a general set of extents by first ordering the extents in ascending order according to their start positions and then ordering those extents with equal

start positions in descending order according to their end positions. Given sets of extents ordered in this way, they discuss the implementation of the containment operators in linear time using a merging strategy. They also provide two explicit reduction functions for conversion of general sets of extents to GC-lists: *to_gcl_min*, which is equivalent to the $\mathcal{G}$ function defined in chapter 3, and *to_gcl_max*, which is defined as follows:

$$\mathit{to\_gcl\_max}(S) = \{a \mid a \in S \text{ and } \nexists\, b \in S \text{ such that } a \sqsubset b\}$$

In contrast to the $\mathcal{G}$ function, the *to_gcl_max* reduction function eliminates extents that are nested within other extents. They give the following properties that relate the two reduction functions and the containment operators for general sets of extents A and B:

$$\begin{aligned} A \rhd B &\equiv A \rhd (\mathit{to\_gcl\_min}(B)) \\ A \lhd B &\equiv A \lhd (\mathit{to\_gcl\_max}(B)) \\ A \not\rhd B &\equiv A \not\rhd (\mathit{to\_gcl\_min}(B)) \\ A \not\lhd B &\equiv A \not\lhd (\mathit{to\_gcl\_max}(B)) \end{aligned}$$

7.3.3 Indirection

No support for structures such as footnotes, links to citations and hypertext is provided by the algebra. A mechanism to support these indirect references might allow the underlying text associated with the solution to one query to be used in the formulation of a second query. Example queries that would be solvable with such a mechanism are

> *Find papers appearing in* JACM *that cite papers containing the sentence "We'll always have Paris."*

or

> *Find WWW pages that refer to Walter Mondale that are referenced from outside North America.*

Several of the query languages described in Chapter 2 provide explicit support for indirection [24, 48, 64, 71, 88].

An extension of the algebra to support indirection might be based on a predicate that compares extents on the basis of the underlying text:

If $a = (p, q)$ and $b = (p', q')$ are extents over the database string $a_1 a_2 ... a_N$ then the predicate $\mathcal{E}(a, b)$ is true if and only if the text strings $a_{\lceil p \rceil} ... a_{\lfloor q \rfloor}$ and $a_{\lceil p' \rceil} ... a_{\lfloor q' \rfloor}$ are equal under an appropriate string comparison.

An indirection operator could then be defined in terms of the predicate as

$$A \ltimes B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } \mathcal{E}(a, b)\}$$

As an example, if W is a GC-list representing World-Wide-Web (WWW) pages, L is a GC-list representing a component of WWW pages that specifies their location, and X is a GC-list representing references to WWW pages, then the query

$$W \rhd ((L \lhd W) \ltimes X)$$

would select those pages in W that are referenced by X.

7.3.4 Co-Existence with the Relational Model

The example on page 33 demonstrates one possibility for representing and searching relational tables. From a different perspective, the structural search algebra could form a basis for extending SQL with text search capabilities. Attributes in a relation could be defined by expressions in the structure algebra. For example, if each tuple in a relation consists of a document identifier, the title of the document, the authors of the document and its abstract, the method of extracting each of these elements from a text database could be specified by the structure algebra.

7.3.5 Syntax Error Analysis

GC-lists may have application in the area of detecting and reporting syntax errors in programming languages, and specifically in the area of noncorrecting syntax error recovery. The basic theory of noncorrecting syntax error recovery for compilers was developed by Richter [86]. His theory had two goals: 1) to allow parser recovery from a syntax error without requiring a heuristic attempt to skip over or repair the error; and 2) to provide a non-ambiguous description of the behavior of the parser when an error is encountered. Unfortunately, he provided no method for implementing his theory.

An implementation for the *bounded context* class of grammars, a subset of the LR(k) class of grammars, was described by Cormack [34]. That method depended on the creation of a substring

parser for a bounded-context grammar. The construction produced a parser that could be applied either right-to-left or left-to-right and allowed the implementation of Richter's theory in linear time for the restricted class of grammars.

A technique for recognizing substrings of LR(k) languages in linear time has been developed by Bates and Lavie [11]. Alone, this technique is not sufficient to implement noncorrecting syntax error recovery for LR(k) grammars. However the data structures described in their paper may be used as a starting point. A further goal is to extend Richter's theory and report the shortest substrings that are not part of any syntactically correct program. The main link with the results of the thesis is through this last property, which is a restatement of the shortest substring search model in the context of syntax error analysis. A further link might be through the "bookkeeping" that would potentially be added to the data structures of Bates and Lavie to develop an algorithm for LR(k) syntax error recovery. It appears likely that this bookkeeping would be similar to that used in the regular expression search algorithm of Figure 5.1.

Bibliography

[1] K. Abrahmson. Generalized string matching. *SIAM Journal on Computing*, 16:1039–1051, 1987.

[2] Alfred V. Aho. Pattern matching in strings. In Ronald V. Book, editor, *Formal Language Theory — Perspectives and Open Problems*, pages 325–344. Academic Press, New York, 1980.

[3] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 256–300. The MIT Press/Elsevier, Cambridge/Amsterdam, 1990.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching — An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[7] Air Transport Association. *Advanced Retrieval Standard — Structured Fulltext Query Language (SFQL)*. ATA 89-9C.SFQL.

[8] Ricardo A. Baeza-Yates. *Efficient Text Searching*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1989.

[9] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. In *16th International Coloquium on Automata, Languages and Programming*, pages 46–62, Stresa, Italy, 1989.

[10] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[11] Joseph Bates and Alon Lavie. Recognizing substrings of LR(k) languages in linear time. *ACM Transactions on Programming Languages and Systems*, 16(3):1051–1077, May 1994.

[12] Elisa Bertino, Fausto Rabitti, and Simon Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

[13] G. E. Blake, M. P. Consens, I. J. Davis, P. Kilpeläinen, E. Kuikka, , P.-Å. Larson, T. Snider, and F. W. Tompa. Text/relational database management systems — Overvew and proposed SQL extensions. Technical Report CS-95-25, University of Waterloo Computer Science Department, June 1995. Supercedes an earlier conference paper [14].

[14] G. E. Blake, M. P. Consens, P. Kilpeläinen, P.-Å. Larson, T. Snider, and F. W. Tompa. Text/relational database management systems — Harmonizing SQL and SGML. In *First International Conference on Applications of Databases*, pages 267–280, Vadstena, Sweden, June 1994.

[15] G. Elizabeth Blake, Tim Bray, and Frank Wm. Tompa. Shortening the OED — Experience with a grammar-defined database. *ACM Transactions on Information Systems*, 10(3):213–232, July 1992.

[16] Günter Born. *The File Formats Handbook*. International Thomson Computer Press, London, 1995.

[17] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[18] M. Bryan. *SGML — An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley, Reading, Massachusetts, 1988.

[19] F. J. Burkowski, C. L. A. Clarke, G. V. Cormack, and R. C. Good. A global search architecture. Technical Report CS-95-12, University of Waterloo Computer Science Department, March 1995.

[20] Forbes J. Burkowski. Surrogate subsets — A free space management strategy for the index of a text retrieval system. In *13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 211–226, Brussels, 1990.

[21] Forbes J. Burkowski. Textriever: A retrieval engine for multimedia databases. In *International Conference on Multimedia Information Systems*, pages 71–76, Singapore, 1991.

[22] Forbes J. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing and Management*, 28(3):333–348, 1992.

[23] N. Charoenkitkarn, M. Chignell, and G. Golovchinsky. Interactive exploration as a formal text retrieval method — How well can interactivity compensate for unsophisticated retrieval algorithms. In *Third Text REtrieval Conference (TREC-3)*, pages 179–199, Gaithersburg, Maryland, November 1994. Proceedings are available electronically [51].

[24] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *ACM SIGMOD International Conference on the Management of Data*, pages 313–324, Minneapolis, Minnesota, May 1994.

[25] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, Atlanta, January 1996.

[26] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-95-40, University of Waterloo Computer Science Department, November 1994.

[27] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[28] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Schema-independent retrieval from hetrogeneous structured text. In *Fourth Annual Symposium on Document Analysis and Information Retrieval*, pages 279–289, Las Vegas, Nevada, April 1995.

[29] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Shortest substring ranking. In *Fourth Text REtrieval Conference (TREC-4)*, Gaithersburg, Maryland, November 1995. Proceedings are available electronically [52].

[30] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(1):377–387, June 1970.

[31] Latha S. Colby and Dirk Van Gucht. A grammar model for databases. Technical Report 282, Indiana University Computer Science Department, Bloomington, Indiana 47405-4101, June 1989.

[32] Mariano P. Consens and Tova Milo. Algebras for querying text regions. In *14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, California, May 1995.

[33] William S. Cooper. Getting beyond Boole. *Information Processing and Management*, 24(3):243–248, 1988.

[34] Gordon V. Cormack. An LR substring parser for noncorrecting syntax error recovery. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 161–169, 1989.

[35] Tuong Dao, Ron Sacks-Davis, and James A. Thom. Indexing structured text for queries on containment relationships. In *Proceedings of the 7th Australasian Database Conference*, Melbourne, January 1996.

[36] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, Massachusetts, sixth edition, 1995.

[37] I. Davis. Adding structured text to SQL/MM part 2: Full text. Technical report, Department of Computer Science, University of Waterloo, February 1996. LHR-24 CAC WG3 N334R2.

[38] Christos Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, March 1985.

[39] Heather Fawcett. *A Text Searching System* — PAT *3.3 User's Guide*. UW Centre for the New Oxford English Dictionary, University of Waterloo, 1989.

[40] M.J. Fisher and M. Patterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings)*, volume 7, pages 113–125. American Mathematical Society, Providence, Rhode Island, 1974.

[41] E. Fox, S. Betrabet, M Koushik, and W. Lee. Extended boolean models. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval — Data Structures and Algorithms*, chapter 15, pages 393–418. Prentice Hall, Englewood Cliffs, NJ, 1992.

[42] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, New York, 1990.

[43] Gaston H. Gonnet. PAT *3.1 — An Efficient Text Searching System — User's Manual*. University of Waterloo, 1987.

[44] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. Lexicographical indices for text — PAT trees vs. PAT arrays. Technical Report OED-91-01, UW Centre for the New Oxford English Dictionary and Text Research, University of Waterloo, February 1991.

[45] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text — PAT trees and PAT arrays. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval — Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice Hall, Englewood Cliffs, NJ, 1992. An earlier version is available as a technical report [44].

[46] Gaston H. Gonnet and Frank Wm. Tompa. Mind your grammar — A new approach to modelling text. In *13th VLDB Conference*, pages 339–346, Brighton, England, 1987.

[47] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeXCompanion*. Addison-Wesley, Reading, Massachusetts, 1994.

[48] Ralf Hartmut Güting, Roberto Zicari, and David M. Choy. An algebra for structured office documents. *ACM Transactions on Office Information Systems*, 7(4):123–157, April 1989.

[49] Marc Gyssens, Jan Paredaens, and Dirk Van Gucht. A grammar-based approach towards unifying hierarchical data models. In *ACM SIGMOD International Conference on Management of Data*, pages 263–272, Portland, Oregon, 1989.

[50] Kathryn A. Hargreaves and Karl Berry. *Regex*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, 1992. `ftp://prep.ai.mit.edu/pub/gnu`.

[51] D. K. Harman, editor. *Third Text REtrieval Conference (TREC-3)*, Gaithersburg, Maryland, November 1994. National Institute of Standards and Technology (NIST), United States Department of Commerce. `http://potomac.ncsl.nist.gov/TREC/t3_proceedings.html`.

[52] D. K. Harman, editor. *Fourth Text REtrieval Conference (TREC-4)*, Gaithersburg, Maryland, November 1995. National Institute of Standards and Technology (NIST), United States Department of Commerce. `http://potomac.ncsl.nist.gov/TREC/t4_proceedings.html`.

[53] Donna Harman. Ranking algorithms. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval — Data Structures and Algorithms*, chapter 14, pages 363–392. Prentice Hall, Englewood Cliffs, NJ, 1992.

[54] Donna Harman. Overview of the first TREC conference. In *16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 36–47, Pittsburgh, PA, June 1993.

[55] David Hawking and Paul Thistlewaite. Searching for meaning with the help of a PADRE. In *Third Text REtrieval Conference (TREC-3)*, pages 257–267, Gaithersburg, Maryland, November 1994. Proceedings are available electronically [51].

[56] David Hawking and Paul Thistlewaite. Proximity operators — So near and yet so far. In *Fourth Text REtrieval Conference (TREC-4)*, Gaithersburg, Maryland, November 1995. Proceedings are available electronically [52].

[57] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[58] R. Nigel Horspool. Practical fast searching in strings. *Software — Practice and Experience*, 10:501–506, 1980.

[59] Institute of Electrical and Electronics Engineers. *Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and Utilities) — Section 2.8 (Regular Expression Notation)*, September 1992. IEEE Std 1003.2.

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

[61] International Standards Organization. *Information Processing — Text and Office Systems — Office Document Architecture (ODA) and Interchange Format*, March 1988. ISO 8613.

[62] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, December 1968.

[63] Brian Kantor and Phil Lapsley. Network news transfer protocol. RFC 977, February 1986. `ftp://nic.ddn.mil/rfc/rfc977.txt`.

[64] Pekka Kilpeläinen and Heikki Mannila. Retrieval from hierarchial texts by partial patterns. In *16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, Pittsburgh, June 1993.

[65] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[66] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[67] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[68] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, 1986.

[69] M. E. Lesk. Lex — A lexical analyzer generator. Computing Science Technical Report 39, A T & T Bell Laboratories, Murray Hill, N. J., 1975.

[70] Arjan Loeffen. Text databases — A survey of text models and systems. *SIGMOD Record*, 23(1):97–106, March 1994.

[71] I. A. Macleod. A query language for retrieving information from hierarchic text structures. *The Computer Journal*, 34(3), 1991.

[72] Ian A. Macleod. Text retrieval and the relational model. *Journal of the American Society for Information Science*, 42(3):155–165, April 1991.

[73] Udi Manber and Ricardo Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, February 1991.

[74] Udi Mander and Gene Myers. Suffix arrays — A new method for on-line string searches. In *First ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, 1990.

[75] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[76] Donald R. Morrison. PATRICIA — Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15:514–534, 1968.

[77] MultiText Project. `http://multitext.uwaterloo.ca`.

[78] Gonzalo Navarro and Ricardo A. Baeza-Yates. A language for queries on structure and contents of textual databases. In *18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 93–101, Seattle, Washington, 1995.

[79] T. Noreault, M. Koll, and M. J. McGill. Automatic ranked output from Boolean searches in SIRE. *Journal of the American Society for Information Science*, 28(6):333–339, November 1977.

[80] J. F. Ossanna. NROFF/TROFF user's manual. Computing Science Technical Report 54, Bell Laboratories, Murray Hill, New Jersey, 1976.

[81] Rob Pike. Structural regular expressions. *Proceedings of the European UNIX User's Group Conference*, 1987.

[82] Rob Pike. The text editor `sam`. *Software — Practice and Experience*, 17(11):813–845, November 1987.

[83] Project Gutenburg. `ftp://mrcnext.cso.uiuc.edu/`.

[84] Tadeusz Radecki. Trends in research on information retrieval — The potential for improvements in conventional Boolean retrieval systems. *Information Processing and Management*, 24(3):219–227, 1988.

[85] D. V. Rama and Padmini Srinivasan. An investigation of content representation using text grammars. *ACM Transactions on Information Systems*, 11(1):51–75, January 1993.

[86] Helmut Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985.

[87] Ron Sacks-Davis, Timothy Arnold-Moore, and Justin Zobel. Database systems for structured documents. In *International Symposium on Advanced Database Technologies and Their Implementation*, pages 272–283, Nara, Japan, October 1994. Invited paper.

[88] Ron Sacks-Davis, Alan Kent, Kotagiri Ramamohanarao, James Thom, and Justin Zobel. Atlas — A nested relational database system for text applications. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):454–470, June 1995.

[89] Airi Salminen and Frank Wm. Tompa. PAT expressions — An algebra for text search. Technical Report OED-92-02, UW Centre for the New Oxford English Dictionary, University of Waterloo, July 1992.

[90] Airi Salminen and Frank Wm. Tompa. PAT expressions — An algebra for text search. *Acta Linguistica Hungarica*, 41:277–306, 1994. An earlier version is available as a technical report [89].

[91] Ari Salminen, Jean Tague-Sutcliffe, and Charles McClellan. From text to hypertext by indexing. *ACM Transactions on Information Systems*, 13(1), January 1995.

[92] Gerard Salton. *Automatic Text Processing — The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, Massachusetts, 1989.

[93] Gerard Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–23, 1988.

[94] Gerard Salton, Edward A. Fox, and Harry Wu. Extended Boolean information retrieval. *Communications of the ACM*, 26(12):1022–1036, December 1983.

[95] Gerard Salton and Michael E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8–36, 1968.

[96] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Computer Science Series. McGraw-Hill, New York, 1983.

[97] Kazem Taghva, Allen Condit, Julie Borsack, and Srinivasulu Erva. Structural markup of OCR generated text. Technical Report 94-02, Information Science Research Institute, University of Nevada, Las Vegas, March 1994.

[98] James A Thom, Justin Zobel, and Bruce Grima. Design of indexes for structured document databases. Technical Report CITRI/TR-95-8, Collaborative Information Technology Research Institute (CITRI), Melbourne, June 1995.

[99] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[100] Frank Wm. Tompa. Not just another database project — Development at UW. In *Tenth Annual Conference of the UW Centre for the New Oxford English Dictionary and Text Research — Reflections on the Future of Text*, pages 82–89, Waterloo, Ontario, October 1994.

[101] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

[102] Ellen M. Voorhees, Narendra K. Gupta, and Ben Johnson-Laird. The collection fusion problem. In *Third Text REtrieval Conference (TREC-3)*, Gaithersburg, Maryland, November 1994. Proceedings are available electronically [51].

[103] W. G. Waller and Donald H. Kraft. A mathematical model of a weighted Boolean retrieval system. *Information Processing and Management*, 15(5):235–245, 1979.

[104] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes — Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.

[105] Gary Wolf. Who owns the law? *Wired*, May 1994.

[106] Sun Wu and Udi Manber. `agrep` —- A fast approximate pattern matching algorithm. In *USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, January 1992.

[107] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.