Using Inverted Lists to Match Structured Text

Sheng Li

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Technical Report CS-96-34[1]

August 30, 1996

[1]This report is a reprint of a thesis that embodies the results of research done in partial fulfillment of the requirements for the degree of Master of Mathematics in Computer Science at the University of Waterloo.

Abstract

Inverted files have long been used as an effective way to implement secondary key retrieval in traditional database systems. For information retrieval systems, postings lists use a similar inverted file structure to accommodate the special characteristics of text data.

This thesis explores extensions to these ideas for structured text databases, especially concerning direct containment of structures. Two kinds of data structures are presented, and for each structure, six basic operations are described and analyzed. Two complex forms of query expressions concerning hierarchical and lateral relationships between text structures are also examined. Examples are presented to demonstrate the effect on performance of each data structure.

Acknowledgements

I am thankful to everyone who supported me throughout the years leading to the completion of this thesis. I would like to thank especially Prof. Frank Wm. Tompa, my supervisor, who gave me much valuable advice and guided me through the difficult times. I also thank Prof. Gordon. V. Cormack and Prof. Ian Munro who served as readers and offered several useful suggestions. I gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Information Technology Research Centre (ITRC) for their financial support. Last but not least, I thank my parents for the encouragement, love, and caring that kept me going all the time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem domain

Text processing, although it has received an increasing amount of attention in recent years, is still mostly handled in an *ad hoc* fashion [BCD+95]. This can be attributed to a lack of attention being paid to the structure of the text. Traditionally, text has been viewed as simply a collection of character strings, and only words in a document are indexed [CCB95]. Now, with the rise of text databases, more and more people are trying to organize text into structured form for ease of use and update.

Conventional database systems have been offering users the means of organizing collections of business data in a structured way. At the same time, they also provide utilities to update, sort, and retrieve stored data conveniently. Combining the power of databases with the versatility of text data would let users maintain the flexibility of string processing, while gaining the advantage of consistency and efficiency of record processing. However, the distinct characteristics of text raise some unique problems for query processing in a text database system.

Unlike conventional data in a relational database, text cannot be well represented in the form of records [GT87]. Instead, text is better stored in variable length unstructured blocks (e.g., paragraphs, chapters). The fact that extensive nesting of data is common in text undermines traditional normal forms in relational databases. In addition, searching can be performed on structured units (or patterns) rather than just words [Kil92, BCD+95]. A simple index on words is sufficient to support many common operations in a

text database. Hence, we need to harness the benefits of existing implicit or explicit document structure to make text processing simpler, more versatile and efficient.

To enhance the text processing ability, many researchers [CCB95, KM93, GT87] have built secondary indexes on those internal structures in textual databases. Ideally, such secondary indexes capture document structure, support efficient structural search on documents, and incur low spatial and computational overhead.

As stated by Clarke, Cormack, and Burkowski [CCB95], one should be able to refer to the structural components of a document when issuing a query in a text database. Therefore, it is important that the secondary indexes can present the user with a well summarized document structure. In addition, we do not want to lose processing speed over the enhanced ability to support structural queries. We present in this thesis a simple variant of inverted indexes that supports efficient structural and content querying. Despite its simplicity, it is extremely powerful.

1.2 Background and related work

Over the years, people have mainly used three types of structure to implement secondary indexes for large structured text. They are, *signature files*, *bitmaps*, and *inverted files*, also known as *posting files* or *concordances* [WMB94].

A *signature* is created by first generating bit patterns on key words of a record using hash functions. The hash values are represented in binary format, and are superimposed, ORed together, to form one bit vector. A signature file is the collection of those signatures to facilitate retrieval [Fal85]. To retrieve a particular record, the bit patterns of all keywords in the query are superimposed, then all records that match this signature are retrieved and scanned to check their validity. The possibility of retrieving and scanning false matches can be decreased by using a longer signature.

Bitmaps, on the other hand, are extreme case of the signature file approach. Each term sets one bit in a signature that is as wide as the number of distinct terms in the collection, and the hash function is one-to-one [WMB94]. This structure is most effective for indexing texts having many words in common, since most of the bits are set in that case.

The most popular choice of implementation is perhaps inverted files. An

inverted file is a type of lexicographical index. It is perhaps the most natural way of indexing a large text, closely resembling the way books are indexed [FBY92, WMB94]. For each term in the given text, sorted in alphabetical order, there is an entry of the inverted file containing a list of pointers in occurrence order to all instances of a particular term in the lexicon, Each term is typically a simple word, but it could be a phrase or any string in the set of documents. Finding all occurrences of a single term in the text requires the scanning of its inverted list entry and following the corresponding pointers [FBY92].

1.3 Problem description & thesis outline

In many text processing applications, an index on words alone is not sufficient. Even for an electronic text, a slightly complicated query, such as finding all entries that were first quoted in a Shakespeare play, relies on concepts such as field or document. Therefore, there is a need for indexes to be built on text units of varied granularity.

The purpose of this research is to design an alternative secondary index structure which is efficient in space and time. Various structural queries that exploit many common relationships among document structures will be used for testing. This will demonstrate the advantages and disadvantages of the new structure over simple lists by comparing them mainly in terms of the processing time.

The next chapter focuses on algorithms for processing several common user queries using simple inverted files. Chapter 3 covers two different inverted file structures and their corresponding algorithms for solving some simple matching problems. We then show some more complex structural queries with some examples and comparisons in Chapter 4. Chapter 5 discusses the implications of using overlapping, instead of non-overlapping, inverted lists for query processing. In the last section, we draw conclusions from the analytical comparisons, and describe some extensions that remain to be investigated. A summary of fundamental definitions, notations, and operations can be found in Appendix A. It is followed by a comprehensive list of algorithms discussed in the body of this thesis along with brief descriptions. Appendix C is a representation of the parent-based partitioned inverted list built on the *Oxford Advanced Learners Dictionary* (OALD) internal tags.

Chapter 2

Conventional inverted file processing

In a conventional database management system, a *data record* or *row in a table* is the unit of information which contains all the pertinent data relating to a unique identifier [DL65]. Records are composed of a collection of related *fields* or *columns*. Fields in such a record describe some specific instance, e.g., a person, an object, or an event [Wie87]. Fields that can uniquely identify a record are called *candidate keys*. One of these candidate keys is usually chosen as *primary key*, which can be used to order the records sequentially for retrieval purposes. For example, in a vehicle licensing division database, the license plate number can be chosen as the primary key because it uniquely identifies each record. Other fields, such as car model, manufacturer, color, year, and principal driver, are secondary attributes. A primary key retrieval would be:

> Give me the record of the vehicle with license plate number XXX 111.

However, in many conventional database applications, primary key retrieval operations alone will not suffice to answer many common queries. For example, in a case of hit and run, police might be looking for:

> A white 1990 Pontiac Grand AM driven by a teenage male.

Here, we need to search on fields such as vehicle make, model, color, registered owner information, etc. Therefore, database management systems

must look for other methods to evaluate this query. These methods search the databases based on values of the fields in records that are not primary keys. These fields are often referred to as "secondary keys" or "attributes" [Knu73], and thus, the name used for such searching is secondary key retrieval.

2.1 Classical inverted lists

One of the most popular techniques of performing secondary key retrieval is to use secondary indexes [AB77]. From as early as the 1950's, one of the important classes of technique to implement secondary indexes for database systems has been the *inverted file* [Knu73, Sta90, CCB95]. The name inverted file is derived from the observation that the role of records and fields are reversed [Knu73]. In conventional databases, an inverted file contains a set of entries, one for each possible value of the indexed field in a relation. Each entry is a list $L(E)$ of pointers to all records that contain a particular term E. The set of lists may be arranged in alphabetical order of the term E in the inverted file to provide rapid access. Each list $L(E)$ has variable length and is maintained in monotonic sequence for efficient logical manipulation. Each such list is referred to as an *inverted list* [Knu73, FBY92].

In this section, we will describe several conventional inverted file processing algorithms and their advantages and disadvantages. This discussion provides the necessary background to comprehend the new structures proposed in this paper.

2.2 Classical inverted file processing

Many models and variants of inverted files have been developed [DL65, Lef69, Lum70, Knu73, C75]. The list processing algorithms handle the following operations [Lef69]:

1. Logical conjunction of nonnegated terms is accomplished by list intersection within the inverted file.

2. Disjunction of nonnegated terms is accomplished by list union within the inverted file.

3. Conjunction of a nonnegated term with a negated term is accomplished by taking the difference of two lists.

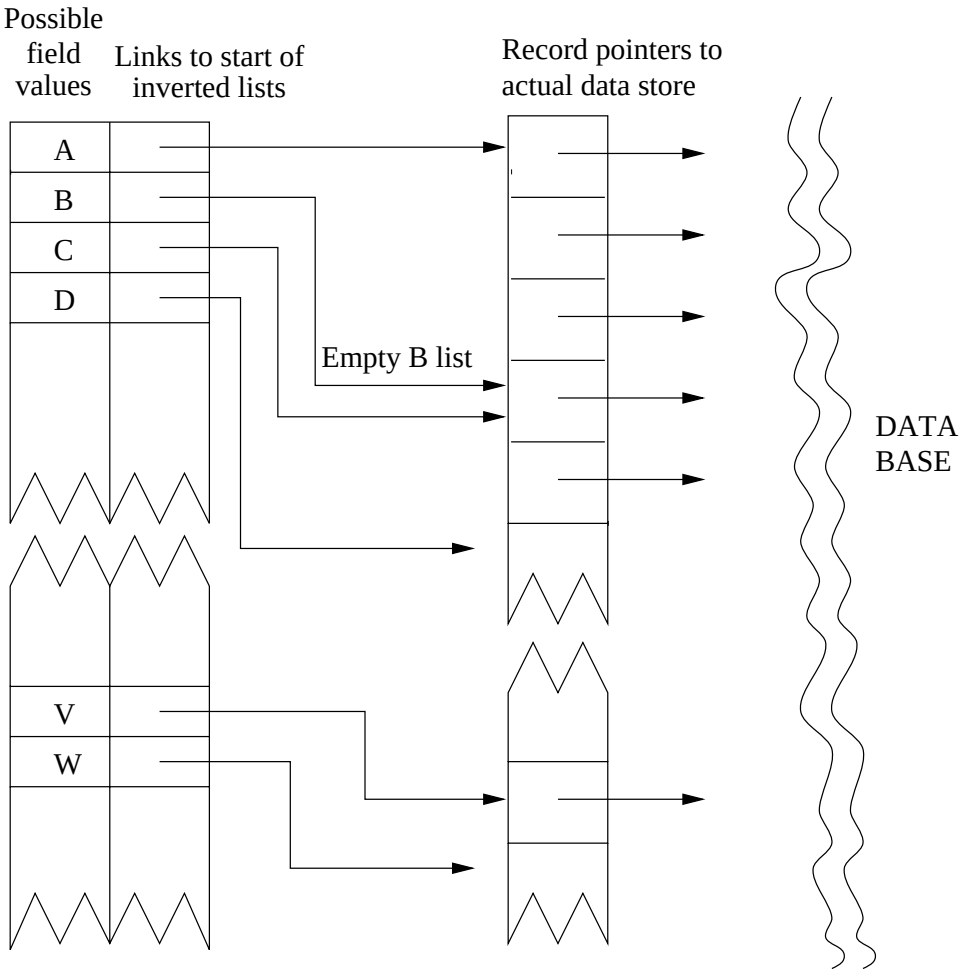


Figure 2.1: Structure of classical inverted file

Merging (*or collating*) means the combination of two or more ordered files into a single ordered file representing the union of the entries from the input file [Knu73]. Merging algorithms were developed by John von Neumann, who first suggested them as early as 1945 [Knu73]. Many specialized structures (e.g., union-find and heaps) have later been developed to make the merging process more efficient. However, the most common merging algorithm is the standard list merging algorithm. All merging algorithms rely on the fact that the initial input lists are already sorted by the same criterion.

Since we deal with lists extensively in our discussions and algorithms, it is necessary to define some common operations that will be applied to them.

nonempty	return true if the list is not empty
headof	retrieve the first element in the list
mark	raise a flag to output the first element later
advance	remove the first element in the list
output	append the first element to the output list, lower the flag if necessary

In addition, give a particular list name, there is an implicit mapping to retrieve the corresponding list. Thus, we will not be concerned with the details of finding a particular inverted lists within an inverted index.

Simple merging of two lists in main memory is the most basic form of list merging. First, the system loads both lists into primary memory and allocates space for the resulting list. The records of both lists are compared, starting from the first ones. We assume, without loss of generality, that the keys are in ascending order. The record with the smaller key gets written to the output buffer and taken off the input list. The next record in that list will be used for the next comparison. This process continues until one of the lists is exhausted. The system then moves the rest of the remaining list into the output merge list as is. Following is a detailed description of the algorithm.

Algorithm 2.2.1 Two-way merge. *This algorithm merges the ordered files* $X = \{x_1 \leq x_2 \leq \cdots \leq x_m\}$ *and* $Y = \{y_1 \leq y_2 \leq \cdots \leq y_n\}$ *into a single sorted file of the form* $z_1 \leq z_2 \leq \cdots \leq z_{m+n}$.

```
TwoWayMerge(X, Y)
  while (X, Y are NONEMPTY)
    if (headof(X) < headof(Y))
```

```
      output X;
      advance X;
    else
      output Y;
      advance Y;
    endif;
  endwhile;
  while (X is NONEMPTY)
    output X;
  endwhile;
  while (Y is NONEMPTY)
    output Y;
  endwhile;
```

This is called an internal merging algorithm and is useful only for small lists. However, database files are generally large in size, and everything cannot be loaded into internal memory at once. In these cases, we have to use an external merging algorithm which utilizes secondary storage to alleviate this space constraint.

For external merging purposes, the algorithm above must be modified so that buffering is used through either virtual memory, which is handled by the operating system automatically, or explicit I/O instructions. At first, only the initial portion of the input lists is loaded into the internal memory, and the space for an output buffer is then allocated. The merge process is performed as usual on loaded data, but when the output buffer is full, it is written out to the external memory to free up the space. Furthermore, the merge input buffers must be repeatedly refilled with the next sequence of items from the same merge file until the entire input is read [Wie87].

Building on the algorithm above, a multiway merging algorithm which works on more than two input lists at the same time can be easily derived. The most obvious way would be to compare records from all lists with each other, again, starting with the lowest ones. The record with the smallest key value is written to the output merge buffer, and deleted from the input list. At any one time, we only need to look at P keys (one from each input buffer) and select the smallest [Knu73]. The process repeats itself until all except one list remains non-empty. The system moves the rest of the remaining list into the output merge buffer.

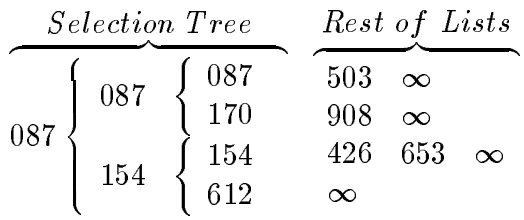
Algorithm 2.2.2 Multiway merge. *$X_1, X_2, \ldots, X_P$ are ordered inverted lists. This algorithm will produce one single inverted list with the same ordering as the Xs as a result of merging all X_i lists.*

MultiwayMerge($X_1, X_2, \ldots, X_P$)
  while *(at least two lists are NONEMPTY)*
    kmin = minimum key value in headof($X_1, X_2, \ldots, X_P$);
    output kmin;
    let *X_k be one list that contains kmin;*
    advance X_k;
  endwhile;
  output the rest of the NONEMPTY list;
end;

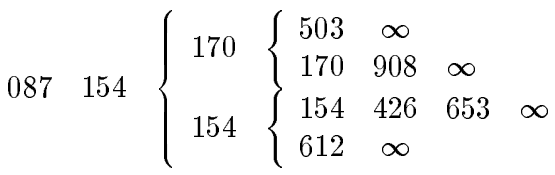
Note that this algorithm keeps all duplicates and places them together in the output list. Eliminating duplicates can be accommodated easily by advancing all lists that contains *kmin* in a loop once a *kmin* is found.

One clever way to speed up the comparison process when the number of lists P becomes large was suggested by Knuth: use a *selection tree* to find the smallest element every time in $O(\log(P))$ instead of $O(P)$ time [Knu73]. For example, consider the case of four-way merging, with a two-level selection tree:

Step 1.



Step 2.



Step 3.

$$087 \quad 154 \quad 170 \quad \left\{ \begin{array}{ll} 170 & \left\{ \begin{array}{lll} 503 & \infty & \\ 170 & 908 & \infty \end{array} \right. \\ 426 & \left\{ \begin{array}{lll} 426 & 653 & \infty \\ 612 & \infty & \end{array} \right. \end{array} \right.$$

$\vdots$

Step 9.

$$087 \quad 154 \quad 170 \quad 426 \quad 503 \quad 612 \quad 653 \quad 908 \quad \infty \quad \left\{ \begin{array}{ll} \infty & \left\{ \begin{array}{l} \infty \\ \infty \end{array} \right. \\ \infty & \left\{ \begin{array}{l} \infty \\ \infty \end{array} \right. \end{array} \right.$$

An additional key "∞" has been placed at the end of each list to allow graceful termination of the merging algorithm, as suggested by Knuth. Since the input lists are usually quite long, adding this terminal record does not significantly increase the length of the data or amount of work involved in the merging. The example also leaves the impression that actual keys and records are moved around in the tree, whereas this can be done using pointers to save on space and computation time. The pseudo-code for tournament selection is provided below.

Algorithm 2.2.3 Selection tree. *A binary tree rooted at T is built such that each of its leaf nodes t_i point to the first element of an inverted list x_i. Note that, in order to simplify the algorithm and ensure it exits gracefully, we append a dummy element with the key of ∞ at the end of each inverted list. The algorithm will merge all elements of inverted lists X_i into one single list, assumed to be in ascending order.*

UpdateTree(R)
  if *(keyof(R) is ∞)*
    return;
  endif;
  if *(R is a leaf node)*
    advance the associated inverted list X_i;

```
    return;
  endif;
  if (LeftChild(R) == R)
    (UpdateTree(LeftChild(R));
  else
    (UpdateTree(RightChild(R));
  endif;
  R = node with the lesser key of (LeftChild(R) and RightChild(R));
end;
```

The above algorithm keeps track of the "winners" (smaller value) of each comparison. There are other methods that can be employed to possibly improve the performance (e.g., parallel list merging [LC88], interpolation [Gon84], tree of losers [Knu73]). However, we will not discuss these structures any further, because either the performance improvement is not significant or additional hardware is required, and the specific algorithm for merging is orthogonal to the thrust of this paper.

Notice we can treat several lists as a single list by adding a few other list ADT operations as follows:

union	multiple lists are merged and the element order is preserved
createvirtuallist	conceptually take the union of several lists to form a single list
list1 − list2	elements of list2 are excluded from list1

For example, if L is a virtual list implemented by a selection tree, then

advance L ≡ UpdateTree(L)

The other operators output, headof, nonempty, and mark perform simple operations that deal with the root node of the tree.

As mentioned above, list intersection can be used to resolve conjunctive queries such as

```
colour = 'white' AND year= '1990' AND make = 'Pontiac'
AND model = 'Grand Am'
```

based on an algorithm very similar to list merging. A two-way intersection on P sorted lists can be accomplished by the following simple process. The algorithm relies, again, on all lists being sorted by the same criterion.

Algorithm 2.2.4 Intersection algorithm

```
Intersection(X, Y)
  while (X, Y are NONEMPTY)
    while (NONEMPTY(Y) and headof(Y) < headof(X))
      advance Y;
    endwhile;
    if (NONEMPTY(Y))
      while (NONEMPTY(X) and headof(X) < headof(Y))
          advance X;
      endwhile;
      while (NONEMPTY(X) and headof(X) = headof(Y))
        output X;
        advance X;
        advance Y;
      endwhile;
    endif;
  endwhile;
  discard all remaining elements;
end;
```

2.3 Partitioned inverted lists

The organization of the simple inverted file creates one and only one inverted list for each possible value of the attribute that has been indexed. There are also other implementations of inverted files that further partition each list.

In the later chapters of his 1969 book [Lef69], Lefkovitz also explained the notion of *cellular inverted list*. The idea is to define logical cellular boundaries throughout the secondary storage into which records may be placed according to some predetermined storage strategy [Lef69]. The conventional inverted lists are partitioned into sublists that will be placed at the beginning of a *cell*. The inverted files contain the addresses of heads of each partitioned list, which also represents the cell addresses that must be searched when evaluating a query.

Consider a query involving the conjunction of attributes with values X and Y. A scan through the top-level inverted file will select only those cells that contain sublists for both X and Y, since one cannot expect to find the intersection of X and Y unless both of those lists are present in a cell. The system will then search for list intersections within the cells selected. The

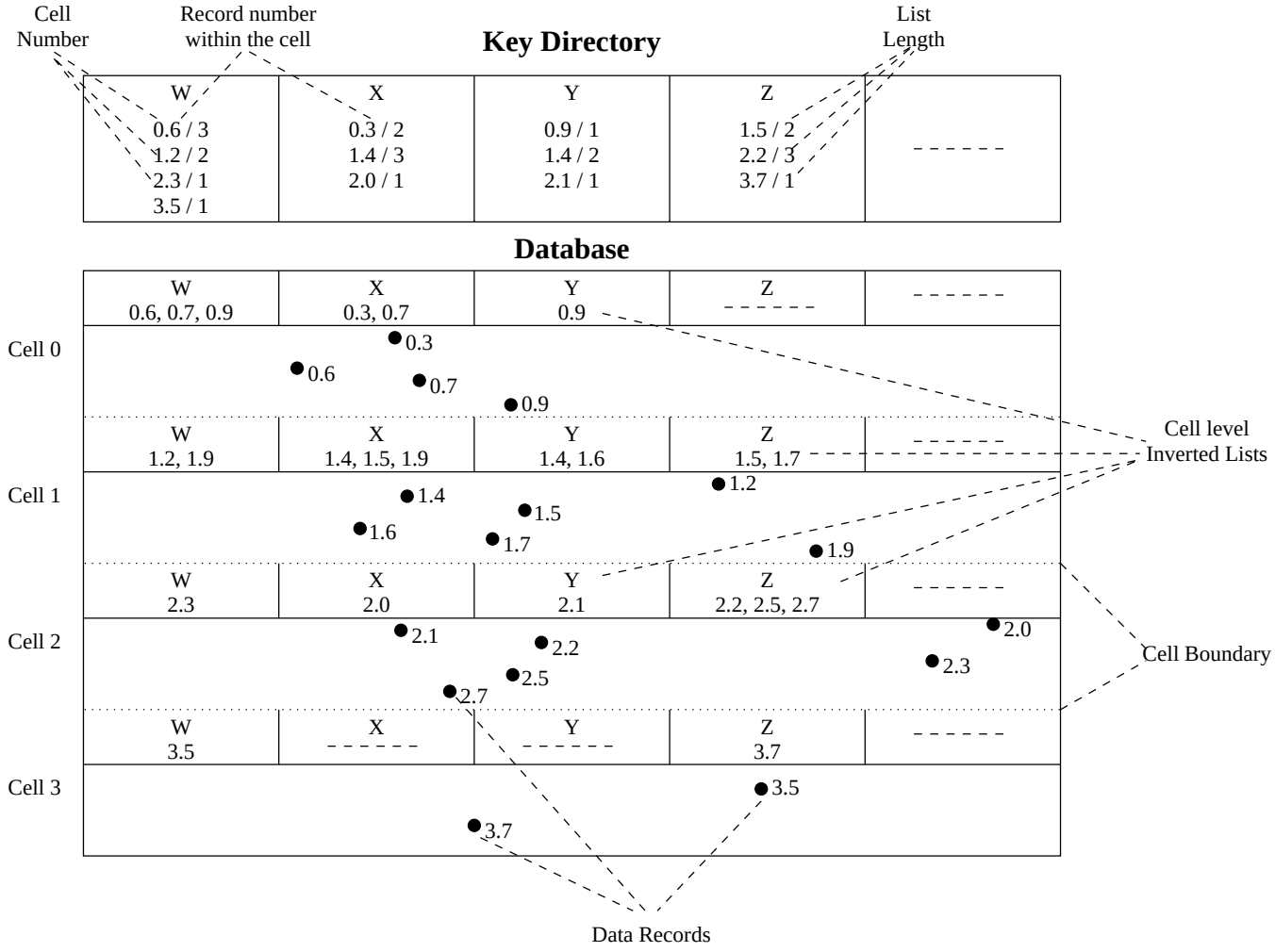


Figure 2.2: Sample organization of a cellular inverted file

advantage of this system is the possible shorter search time of the inverted lists. A number of cells/sublists are likely to be eliminated before any inverted lists are accessed, and shorter lists will terminate the intersection algorithm sooner.

A more generalized description of the partitioned inverted file organization was published by Alfonso F. Cárdenas in 1975(Figure 2.3). Cárdenas proposed dividing everything into four levels of blocks: *data blocks, record address blocks, key-value index blocks, and a track index block.* Each block is the unit of transfer between secondary and primary memory as a result of a single I/O operation [C75]:

1. *Data blocks* are complete records in the database including the key values.

2. *Record address blocks* are accession blocks which contain lists of pointers to actual records. Each list is ordered by pointer values.

3. *Key-value index blocks* are the inverted list headers. Each entry of the list contains an *access key*, which is the key name/key value pair. In other words, the entries are ordered by key name, then by key value. Each entry also contains a pointer to one of the lists in the record address block, along with a list length.

4. *Track index block* provides an easy way to reach the beginning of each block in the key-value index blocks. The entries have the same format as the key-value index blocks. Each contains the access key value that appears at the top of each key-value block and the pointers to these blocks.

Cárdenas' approach treats the inverted file index itself as a database problem. The track index block is a simple yet effective index built on top of the inverted file, and the inverted lists are partitioned by the physical transfer unit – blocks. The database engine can quickly jump to the beginning of the block where the first entry of a key name/key value pair occurs. This arrangement reduces the time required to search through irrelevant index records, which can be substantial when the lists are huge.

The above mentioned cellular and partitioned inverted lists have an interesting property: they are both divided according to the record placement in secondary storage. This is interesting because partitioning according to

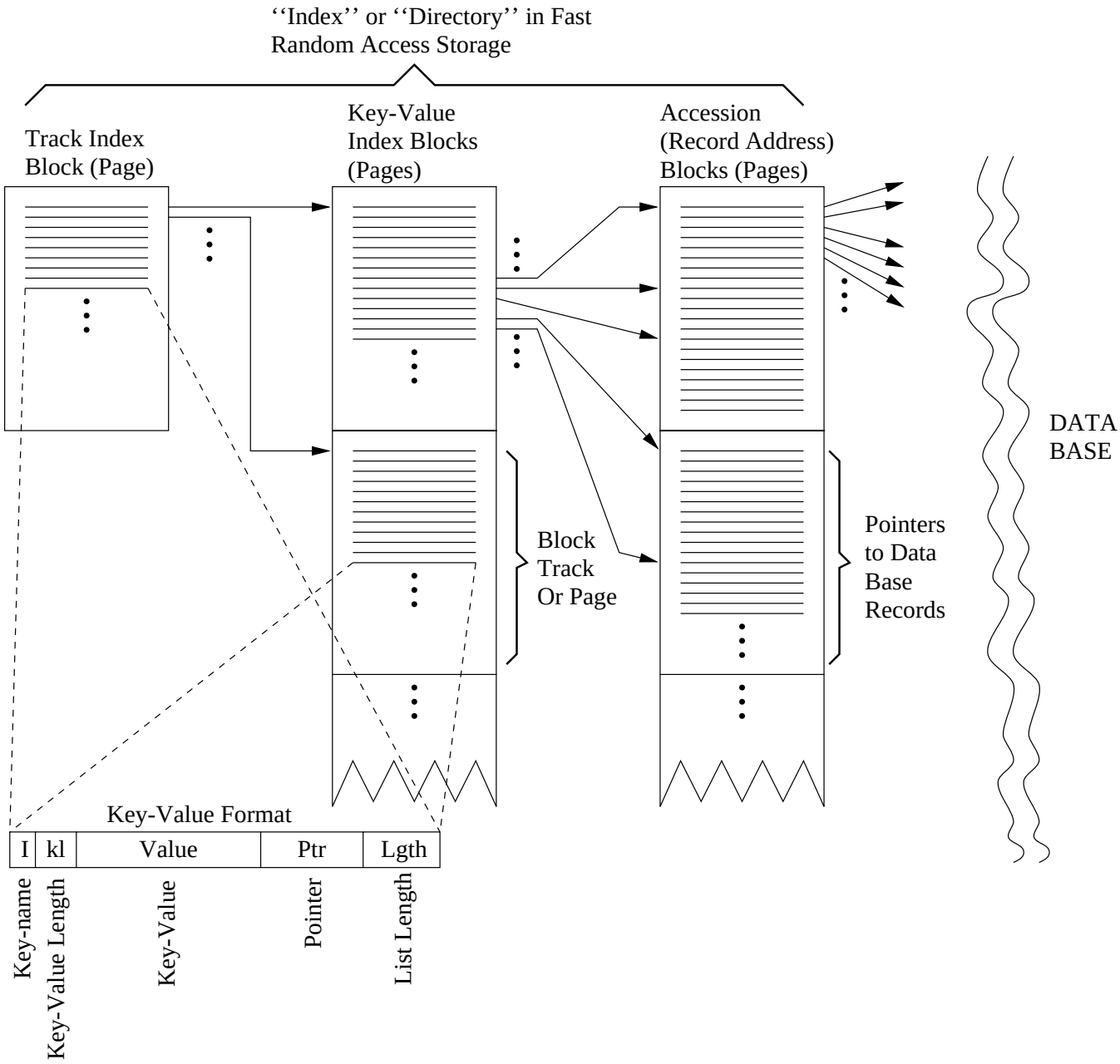


Figure 2.3: Multilevel physical layout for the inverted file [C75]

physical boundaries determines another important property for partitioned lists. That is, any sublists of an attribute value X can only merge with at most one sublist of another attribute value Y. That is why merging with inverted lists performs so much better than arbitrary relational *joins* where each record of X must be compared with every record of Y.

All the algorithms above are well suited for supporting selection from relational databases where a well-defined and well-recognized structure exists among data. Modifications are needed for text databases, as we will see in the next chapter.

Chapter 3

Simple structure matching

Inverted files have been popular for text databases as well as relational databases [Sal68, Sal75, Ger83, WMB94, Tom89, KLMN90, Sta90, BBT92, FBY92, ST93, CCB95]. An inverted file closely resembles the way books are indexed. The format of an inverted file changes slightly when used in a text database, mainly because text databases lack the kind of explicit structures that are present in conventional databases. Indexes reference lexical items instead of records. An indexed term together with its referents in a text database is sometimes called a *postings list*, while each referent is a *posting*; thus, inverted files are commonly referred to as postings files. Each term is typically a simple word, but it can also be a phrase or any arbitrary length string in the set of documents [SM83, WMB94]. The references to the actual occurrences of terms can point to the term itself, or to fields or documents that contain it. In the case where the text consists of a set of independent documents and all postings reference documents as a whole, a postings file operates similarly to a conventional inverted file (i.e., documents play the role of records). If documents can overlap one another or postings refer to subfields of documents, then the processing of the postings lists must be modified from conventional inverted file algorithms.

3.1 Assumptions and definitions

Before we start investigating the modifications necessary for text processing, we state a few assumptions on which this paper is based and some definitions for terms and notations that will be used.

A *text segment* or *region* is a predefined contiguous piece of text of arbitrary length.

The type of a text segment is a predefined role that this segment plays in the whole document or set of documents according to some specification (e.g., a grammar) [ST96]. For example, *paragraph* can be the type of some text segments in a document.

We assume there is a known, finite collection of unique text segment types $x_1, x_2, \cdots, x_n$. The set of possible text types forms a *universal set* $\mathcal{S}$. In other words, $\mathcal{S} = (x_1, x_2, \ldots, x_n)$. New text types can be added to the universal set only after they are defined, the text is scanned, and indexes are updated. From this point on, when we mention text segments in the following discussion, it is always with respect to this universal set.

Since the main purpose of this paper is to discuss query performance, we will assume that indexes using the various structures under examination have been built; index building techniques will not be discussed in this paper. Some examples of modern text index building methods can be found in [Wie87, GBYS92, WMB94].

One text segment is said to contain another if the latter begins at or after the start of the former and ends before or at the end of the former. When a text segment A contains a distinct text segment B, and there does not exist a distinct text segment C that contains B and is contained in A, we say that text segment A *directly contains* text segment B [ST96]. Because of its similarity to a Parent-Child relationship in a tree structure, we refer to that text segment A as the *parent* of text segment B. Conversely, we refer to text segment B as the *child* of text segment A. If the existence of text segment C cannot be determined or is immaterial, then we simply say that text segment A *contains* text segment B. Finally, when text segment A appears before segment B without overlap in a document, we say A is *followed by* B (regardless of whether other segments also follow A and are followed by B). A summary of fundamental definitions, notations, and operations can be found in Appendix A.

For the body of this thesis, we assume that two regions are either disjoint or properly nested with respect to each other. That is, the collection of all regions in a text is assumed to form a tree structure. This assumption is relaxed in Chapter 5.

3.2 Exploitation of text structure

To improve text retrieval, users need to exploit the implicit structures present within the text. Loeffen [Loe95] and Baeza-Yates and Novarro [BYN96] provide extensive surveys of models to retrieve both contents and structures from text. The PAT system, which was used to construct the online *Oxford English Dictionary* at the University of Waterloo as well as serving as the basis of the Open Text Index, one of the search engines on the World Wide Web, relies on the notion of *region* to handle structure queries [Tom89, ST93, BCD+95]. A region is defined by a pair of references to the start and the end of a contiguous segment of text of arbitrary length. All pairs of references to the same type of text segment (e.g., paragraphs) are stored in a single list. Therefore, there are as many such lists in the system as there are types of text fragments identified by the text design. For example, in documents defined by an application of the Standard Generalized Markup Language (SGML) [Gol90], there may be a region list for each element type identified in the document type descriptor.

To illustrate, consider a collection of email messages(Figure 3.1). Region lists can be built on various header fields, such as senders, and on the body of the messages, such as paragraphs. Furthermore, the presence of region lists is in addition to the existing postings lists.

As a baseline, we will examine a structure similar to the standard inverted list, which will index regions (fields) as well as postings for words or phrases. Just like the standard inverted lists, the new lists are sorted in monotonically increasing order of the occurrences of the regions in the indexed text.

For simplicity's sake, we will refer to the conventional lists for regions as simple inverted lists from this point on.

Since we will operate in terms of regions and region lists in our discussion, it is necessary to define a few operations that will be used on those abstract data types.

The following notations are first introduced in the MultiText project [CCB94]. Elements a and b are particular regions each containing a pair of references (s, e) to the start and end points of the underlying text, we define:

$a \rhd b$ to mean a *contains* b: $\{a.s \leq b.s$ and $a.e \geq b.e\}$

$a \lhd b$ to mean a *contained by* b: $\{a.s \geq b.s$ and $a.e \leq b.e\}$

a) Schematic showing pointer pairs for regions of type R

Offset	Text
000	**Date:** Fri, 26 May 1996 11:50:20-0500 (EST)
045	**From:** Patrick Li <s3li@uwaterloo.ca>
081	**To:** Do-it-all@auto.dealer.com
111	**Subject:** Buying a car
134	
135	*Hi,*
139	
140	*I am referred by one of your customers, Frank, who told me that you*
207	*have a large selection of automobiles and very competitive prices.*
274	
275	*I'm looking for a small and economical vehicle in the price range of*
342	*$14,000 - $15,000. I prefer a used car with about 2 yrs/30,000 km*
409	*on it. Standard transmission would be nice, but air conditioning*
474	*and stereo cassette are must!*
503	
504	*If you have something that fits my description, please let me know.*
572	
573	*Regards*
581	
582	
583	
584	
585	**From:** Do-it-all@auto.dealer.com
617	**Date:** Fri, 26 May 1996 14:10:30-0500 (EST)
660	**To:** Patrick Li <s3li@uwaterloo.ca>
696	**Subject:** Re: Buying a car
722	
723	*I have exactly what you are looking for. Why don't you come down*
789	*here and take a look for yourself.*
⋮	⋮

b) Sample region vectors using byte offsets

Figure 3.1: Illustration of region lists combined with posting lists

	Return Type		
	Inner	Outer	Both
General Containment	A..D#	A#..D	A#..D#
Direct Containment	P.C#	P#.C	P#.C#

Table 3.1: Categorized basic containment operations

$a \not\rhd b$ to mean a does *not contain* b: $\{a.s > b.s$ or $a.e < b.e\}$

$a \not\lhd b$ to mean a is *not contained by* b: $\{a.s < b.s$ or $a.e > b.e\}$

We also introduce two additional operators that are simple extensions to those defined for the MultiText Project.

$a \prec b$ to mean a *starts before* b: $\{a.s < b.s\}$

$a \dashv b$ to mean a *ends before* b: $\{a.e < b.e\}$

As before, given a particular region type name, there is an implicit mapping to retrieve the corresponding region list containing all instances of that region type.

3.3 Basic operations & algorithms

The basic operations and accompanying algorithms supported in all systems to be examined are shown below. Queries involve containment of descendents (D) within ancestors (A) or direct containment of children (C) within parents (P); they also specify which of the qualifying text segments to return.

Following syntax proposed earlier [BCK+94], we define six variants as shown in Table 3.1: Query A..D# asks the engine to find all instances of descendent region type D that appears *anywhere*, directly or indirectly, inside text type A. Query A#..D is very similar, except that we are interested in the A*s* instead of the D*s*; and for A#..D#, we are interested in regions of

both types. Alternatively, P.C# specifies a request to find all instances of region type C directly contained within a region of type P. Notably, PAT and the MultiText system support only forms of containment returning a single type and not requiring direct containment (i.e., A..D# or A#..D only).

We shall now examine how inverted list structures can support the above queries.

3.3.1 Simple inverted lists

If region types are represented by postings lists and we wish to evaluate containment queries (e.g., A..D#), it is obvious that list merging of some sort must be employed. The list merging algorithm used here is based on the conventional ones mentioned in the previous section with one major modification to suit the need of text. During the comparison of the top elements in each list, the system will be looking for *region containment* instead of *point equality*.

Algorithm 3.3.1 depicts the procedure of merging two ordered inverted lists, retaining all regions that satisfy the containment relations:

Algorithm 3.3.1 Generalized two-way intersection for text regions. *For any two ordered inverted lists of regions* $X = \{x_1, x_2, \ldots, x_m\}$ *where* $x_1 \leq x_2 \leq \cdots \leq x_m$ *and* $Y = \{y_1, y_2, \ldots, y_n\}$ *where* $y_1 \leq y_2 \leq \cdots \leq y_n$, *the algorithm will produce two similarly ordered inverted lists that are either shorter or equal in length. Each element of all lists is a pair of reference pointers* (s, e) *that defines the boundaries of a region.*

```
RegionMerge(X, Y)
  while (both X and Y are NONEMPTY)
    while (NONEMPTY(Y) and headof(Y) ≺ headof(X))
      advance Y;
    endwhile;
    while (NONEMPTY(Y) and headof(X) ▷ headof(Y))
      output Y;
      advance Y;
      mark X;
    endwhile;
    if (X is marked)
      output X;
```

```
    endif;
    if (NONEMPTY(Y))
      while (NONEMPTY(X) and headof(X) ⊣ headof(Y))
        advance X;
      endwhile;
    endif;
  endwhile;
  discard all remaining elements;
end;
```

Note that similar to the two way intersection (Page 16), this algorithm also discards the list that has not been exhausted, because the remaining elements cannot contain nor be contained by a region of the other type.

To solve general containment problems of the form A..D, a system using the simple inverted file organization would apply RegionMerge to the inverted lists for regions A and D. This algorithm takes $O(|A|+|D|)$ time to compute, where $|X|$ is the cardinality of the inverted list for text type X.

The general merge algorithm keeps both the A list (ancestor) and the D list (descendent). If we are only interested in the contained regions (i.e., elements of D list), we need not *mark* and *output* X. If we are only interested in containing regions (i.e., elements of the A list), then we need not *output* Y.

Although neither PAT nor the MultiText system supports the variant of this operation that maintains both lists (i.e., A#..D#), it can be simulated in those systems by issuing the other two operators sequentially.

The problem of direct containment (i.e., P.C#) is significantly more difficult. The simple inverted list structure provides little assistance to such queries, because it does not distinguish direct from general containment. First, the engine can use Algorithm 3.3.1 to find all regions that satisfy the general containment P#..C#. Then, for each instance of C in the intermediate result, either a string search in the actual text must be used (assuming there is sufficient mark-up in the text to identify all regions that could contain regions of type C) or all indexes for every other text type must be examined to ensure that no other region containing that C text region is also nested within the corresponding P region. For large data collections, either approach would result in significantly increased I/O operations caused by the loading of possibly numerous text segments, or even all of the index files, scattered over the secondary storage.

The following algorithm formally describes the second approach where all other region indexes are used in a filtering process to guarantee direct containment. The two lists of interest are repeatedly compared against each list of some other type, eliminating elements that are found to be not in direct containment. (N.B. Since self-nesting of region types is not permitted, we need not check for interceding regions of type P or of type C.)

Algorithm 3.3.2 Filtering. *Given the inverted lists* $X = x_1 \cdots x_m$ *and* $Y = y_1 \cdots y_n$ *be the output of Algorithm 3.3.1 when applied to types P and C. This algorithm will compare those lists against every other list in the system and produce two lists less or equal in size to X and Y, but all elements of X directly contains some elements of Y and all element of Y is directly contained by some element of X. L is the set of all inverted files that are defined over the universal set S.*

```
Filter (X, Y)
  forall (lists W ∈ {L − {P, C}})
    while (X, W, Y are NONEMPTY)
      while (NONEMPTY(W) and headof(W) ≺ headof(X))
        advance W;
      endwhile;
      while (W, Y are NONEMPTY and headof(X) ▷ headof(Y))
        if (headof(X) ▷ headof(W))
          while (NONEMPTY(Y) and headof(Y) ≺ headof(W))
            output Y;
            advance Y;
            mark X;
          endwhile;
          while (NONEMPTY(Y) and headof(W) ▷ headof(Y))
            advance Y;
          endwhile;
          while (NONEMPTY(W) and headof(W) ⊣ headof(Y))
            advance W;
          endwhile;
        else
          while (NONEMPTY(Y) and headof(X) ▷ headof(Y))
            output Y;
            advance Y;
```

```
            mark X;
          endwhile;
        endif;
      endwhile;
      if (X is marked)
        output X;
      endif;
      advance X;
    endwhile;
    X := output list for X;
    Y := output list for Y;
  endfor;
end;

FindDirect (P, C)
  RegionMerge(P, C);
  Filter(P, C);
end;
```

As we discussed before, the RegionMerge() function take $O(|P| + |C|)$ time, where $|X|$ is the cardinality of the inverted list for text type X. The Filter() function on the other hand, loops through all other inverted lists for any region other than P and C and compares each one with the results produced by RegionMerge(). Thus, the time it takes is $\sum_{l_i \neq P,C}(|l_i|+|P|+|C|)$ where $l_i \in L$. Assuming each list size is bounded by some constant n, then, in the worst case, the function takes $O(n \times |L|)$ time to compute, where $|L|$ is the number of region types.

3.3.2 Region lists partitioned by parent type

It is clear that one of the main problems inherent in simple inverted lists is the lack of ability to distinguish direct from general containment. There is no explicit information concerning the hierarchical relationships between structural units. One approach is to avoid the problem, as proposed by Consens and Milo [CM94]. Wherever possible, they reduce direct containment operations into general containments with the additional knowledge of internal document structure. However, this approach only works under certain

circumstances (e.g., when region A is constrained to be contained and only contained by region B).

Here, we present a new data structure based on simple inverted lists to handle the direct containment problem efficiently, while introducing little overhead in space and in the time needed to answer general containment queries.

Intuitively, the easiest way to solve the direct containment problem is to set up a link from each parent region to its children (like a tree). However, that would mean at least two additional pointers to every entry in the inverted file; one for the first child and another for the next sibling. This would virtually double the primary and secondary memory requirement of the index structure. Consequently, this added demand will slow down response time to queries because of increased input/output operations. On top of this, the intersection would still be costly, because of the need to traverse all the appropriate children links during retrieval. Such a drastic performance hit is unacceptable; we must avoid incurring too much space overhead while adding extra structural information.

Instead, we will partition the inverted lists and reorder them in such a way that finding the children of a particular region type is almost as easy as finding the region itself. By reordering, we only have to keep track of the bounds of each sublist without creating huge spatial and computational overhead.

The new structure as depicted in Figure 3.2 can be described in the following way:

1. A set of inverted lists for text region types.

2. A simple partition of those inverted lists (with references to division points) based on the type of the *parent*, where elements in each partition is ordered by position.

Each inverted list contains references to all instances of a region type. Furthermore, within those lists, the elements are sorted in a predefined (e.g., alphabetical) order by the type of their parent. This is possible because of the assumption of strict hierarchical nesting. However, the important distinction between this structure and the simple inverted lists is that each sublist(group) can be accessed as an independent unit, while loading the complete list for any region type is just as easy as before, since all partitioned inverted lists are stored contiguously on physical storage.

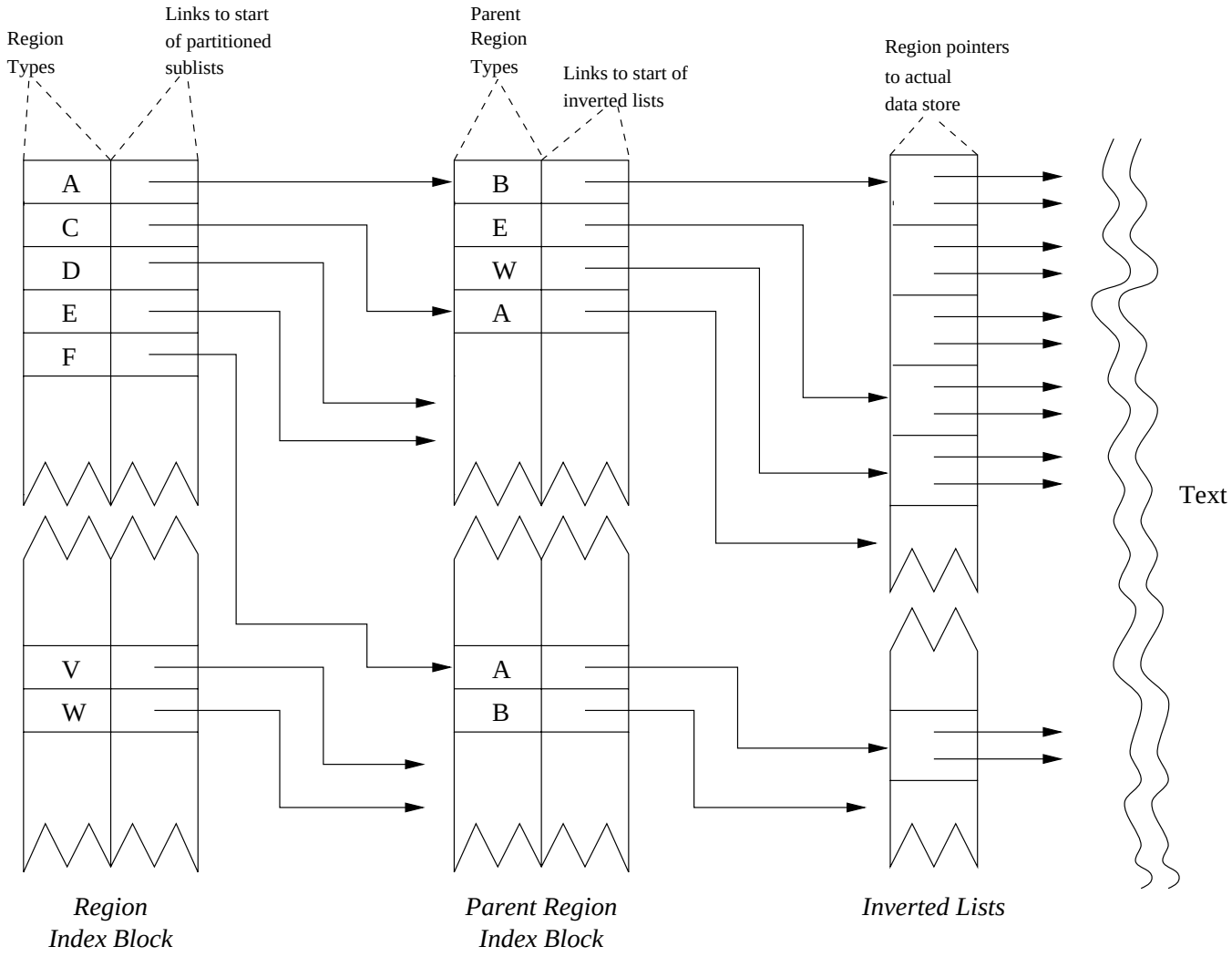


Figure 3.2: Structure of inverted lists partitioned by parent region type

With this arrangement, one can easily answer queries represented by expression P.C#, because all qualified regions are already grouped into one ordered inverted list. Only two pointers (for C, then P) must be traversed to locate the result set.

Using this structure, the processing of query P#.C or P#.C#, however, is not as easy. Since we only have the type of the parent entry but not the reference to the parent itself, it is not possible to retrieve this information with one step. Storing a reference to the parent region is feasible, but requires additional space in the index. Instead, we could use the algorithm that was applied to the simple inverted list (i.e., finding both P and C lists, and then merge them in a similar fashion as with the simple inverted lists). The only complication is that there could exist multiple lists for P, since it too is partitioned by parent. However, we can utilize the createvirtuallist operation defined earlier (page 16) to hide the fact that there is more than one list for P. Since we now conceptually form a single list for P, and there exists at most one list for C*s* directly contained by P, the conventional two-way region merge can be performed on them as before.

Algorithm 3.3.3 Finding Parent and Child. *Given two region types P and C. This algorithm finds all P regions that directly contains C. Both P and C are kept in the answer set. X is a reference to an inverted list.* Find*() is a simple operation that retrieves an inverted list that corresponds to its parameter.*

FindBoth (P, C)
  X := Find(P.C);
  RegionMerge(createvirtuallist(all lists for P), X);
end*;*

Assuming that the virtual list is implemented using the selection tree algorithm discussed before, the run time of this algorithm is $O(I + \tau \log(J + 1) + |X|)$, where $\tau = \sum_{\forall i} |P_i|$. In other words, τ is the sum of the sizes of all partitioned lists for region type P. I is the constant time to traverse the pointers to find P.C, $|X|$ is the size of the list X, and J is the total number of input lists that formed the virtual list. In this case, J is the number of sublists for P. The $\log(J + 1)$ part of the formula is derived from updating of the *selection tree*.

In the worst case analysis, we know the upper bound of any complete region list is n, therefore, τ is $O(n)$. Furthermore, $|X|$ is also $O(n)$. Of

course, if P can be contained by every other region type in the text, $J+1$ is $O(l)$, the total number of region types. Finally, we note that I is $O(1)$. Thus, the formula can be simplified to $O(n\log(l))$.

Like the simple inverted lists, we need only remove *output Y* (i.e., the child region list) from RegionMerge if we are only interested in solving P#.C. Since the removal of one output step does not change the computation complexity, the analytical time will remain the same.

To answer queries represented by expression A..D#, the engine will first determine the result set of A.D#, since it is a part of the answer, and more importantly, it can be retrieved with very little effort. Secondly, the engine must retrieve all other partitioned inverted lists for text type D (i.e., lists that do not have A as their parent), and all of the inverted lists for region A. Thirdly, two virtual lists will be created, one for region A and another for region D, using the createvirtuallist operation. Then, merge those two lists together (as for simple inverted lists) using the modified two-way merge that only keeps the children. Lastly, take the union of the list for A.D# with the list produced by the merge to arrive at the correct result.

Algorithm 3.3.4 Find Descendent. *X is an entry point to an inverted list.*

FindDescendent (A, D)
  X := Find(A.D);
  Y := RegionMerge(createvirtuallist(all lists for A), createvirtuallist(all lists for D - X));
  return union(X,Y);
end;

The run time of this algorithm is $O(\tau\log(J+1)+\eta\log(K)+I)$, where I is the time taken to retrieve A.D, $\tau=\sum_{\forall i}|A_i|$, $\eta=\sum_{\forall i}|D_i|$, J is the total number of partitioned inverted lists for A, and K is the total number of partitioned inverted lists for D. Again, using the same reasoning as above (Algorithm 3.3.3), we see that the time is $O(n\log(l))$.

Intuitively, one can see that the process to solve query A#..D and A#..D# would be very similar to query A..D#. In these cases, we cannot merely prefetch A.D, so we resort to what is essentially the algorithm for simple inverted lists: two virtual lists built from all sublists of A and sublists of D will be used in the two-way merge. With query A#..D#, no modification to RegionMerge is needed since we do need both the ancestor and the descendent in the answer. On the other hand, query A#..D is only interested in

the ancestor, therefore, removing the code to *output Y* (i.e., the child region list) from RegionMerge is necessary.

Algorithm 3.3.5 Find Ancestor and descendent. *This function is used to find two sets of regions of type A and D that satisfy the condition* $\{a \rhd d \mid a \in A, d \in D\}$.

FindAD (A, D)
RegionMerge(createvirtuallist(all lists for A), createvirtuallist(all lists for D);
end;

Clearly, the computational cost of this algorithm is again $O(n\log(l)$ because the computationally intensive part of the algorithm is exactly the same as the previous algorithm.

3.3.3 Analysis and critique

Looking at the worst case analytical results of the algorithms for both simple inverted lists and parent-based partitioned inverted lists, we can make several observations. First of all, each structure performs better than the other under certain circumstances. More specifically, the simple inverted lists structure shows its strength in resolving *general containment* queries (i.e., A..D), while the partitioned inverted lists wins hands-down for *direct containment* type of queries (i.e., P.C).

Following is a summary of analytical computation time between the simple inverted lists and the partitioned inverted lists. We will review the results in the subsequent paragraphs.

How much better the simple inverted list is for general containment queries depends on the total number of possible parents that each region type may have. The fewer the possible parents, the better the partitioned inverted list will perform. If most of the queries concentrate on region types with a small number of parents (e.g., one or two) then both structures will have similar performance. Note that the input/output cost for solving these queries need not increase because all partitioned inverted lists can be stored together on secondary storage. Once we find the address of the first sublist for any region, all lists can be loaded into main memory in blocks.

However, for direct containment, the partitioned inverted lists would significantly out-perform the simple model. With the simple inverted list structure, the engine needs to merge almost every list in the database system

Sample Queries	Simple Inverted Lists	Partitioned Inverted Lists
A..D#	***O(n)***	*O(nlog(l))*
A#..D	***O(n)***	*O(nlog(l))*
A#..D#	***O(n)***	*O(nlog(l))*
P.C#	*O(nl)*	***O(1)***
P#.C	*O(nl)*	***O(nlog(l))***
P#.C#	*O(nl)*	***O(nlog(l))***

n =upper bound of unpartitioned inverted list size
l = total number of region types

Table 3.2: Worst case computation times for basic containment

against the lists produced by performing P#..C#, regardless of whether the parent or the child or both region types are requested by the user and regardless of the number of parent types for any given type in the database.

With the partitioned inverted list structure, the engine can take full advantage of the extra hierarchical information to speed up query processing. First of all, the computation cost for P.C# is constant, because the engine needs only one access to the *parent region index block* to find the exact location and size of the partitioned inverted list for P.C#. If the query requires identifying the parent regions, however, the worst possible scenario is that all instances of the child region occur within the parent type but could have any other region types come in between them.

There is a special case, however, where finding the parent regions only takes a constant time: when the answer to the query is empty. In this case, no children would satisfy the criteria and thus the search can be halted

without list merging. This is impossible to achieve for the simple inverted lists because no such hierarchical information is built into the index.

Note that the worst case computation time for partitioned inverted lists for most queries are the same. This is because multiway list merge and selection trees are employed in all those cases.

In terms of space requirements, naturally, the partitioned inverted list will require more than the simple inverted list. The extra information, *child region index block*, will take at most l^2 units of space. Each unit is the offset into the actual inverted file where the first entry in the sublist is stored. In general, the total number of distinct region types will not be large. Even applications as complicated as the electronic *Oxford English Dictionary*, have only fifty to seventy different tag types. Assuming each offset takes about four bytes, the whole table would take up around 4k of memory or about 1 page in many systems. With the size of today's primary and secondary storage media, this is not of any burden. In fact, the system should be able to load this table into the main memory at start up time and keep it there for the duration of the program.

Another extra requirement that the partitioned inverted lists demand is the creation of the *parent/child index block* at index building time. If the regions are based on tags in the text, such as SGML, the information required to build the blocks should be readily available during the scanning of the text. Therefore, all that is added is the allocation of a buffer in the main memory for building the indexes and in the secondary storage for a permanent copy. During the creation, the actual inverted list records along with their parent type is written to the secondary storage in the order that they appear in the document when scanning. After the whole text is scanned, those lists will then have to be read back into the system, again, one list at a time, to reorganize them according to the parent. The records will then be written back to the secondary storage in blocks, ready for use. Thus, in terms of computation time, every region record will be read in and written out once more than for the simple inverted lists. However, this is only a one-time cost, which is a small price to pay compared to the time that this will save during retrieval.

Chapter 4

Complex structure matching

4.1 Matching a chain

We have defined the basic operations in the previous section involving two region types. These operations may suffice for the requirement of a simple search engine, but in a more sophisticated application, users are likely to ask questions about more than two nested regions at once. Thus, it is important to find out how differently the simple and partitioned inverted lists will behave under the increased complexity in queries. We call such queries with three or more attributes a *chain*.

Let us consider a slightly more complicated form of query:

$$A\#..D\#.C\# \tag{4.1}$$

This query describes a piece of text such that there is an instance of region D that is the parent of some C region and is contained in an instance of A. All collections of qualified As, Ds, and Cs are to be kept as the answer.

We have requested all three regions be kept in the result set because from our experience in the previous section, the cost of determining one region instead of two, with the exception of finding the child in the case of the partitioned inverted lists, is the same under most situations for both structures.

4.1.1 Simple inverted list

There are two approaches to process this query: iteratively apply each operator or perform the multiway merge on all three attributes at the same time.

The exact procedure is not as simple as it sounds. Because of the direct containment relationship in D.C, we have to filter out all the irrelevant regions by comparing the C and D lists against all other region lists. This process should not be done during the merge step because otherwise all inverted lists in the system need to be loaded and compared against each potentially valid A..D pair. That will cause excessive I/O if all lists cannot fit into the memory simultaneously, which is fairly common in practice. Thus, the direct containment query should be resolved before the multiway merge.

The merge process is an extension to the two attribute RegionMerge() algorithm. If the direct containment relationships are already resolved, it will only need to worry about general containments. The algorithm takes first two successive lists in a chain, X and Y, and finds a matching pair using the technique shown in two way RegionMerge(). It then descends the chain using Y and its descendent as the parameter. If a match is found, then the program calls itself again, until the end of the chain is reached or no valid descendents are found. The algorithm will terminate when one of the lists is empty.

Algorithm 4.1.1 Recursive multiway merge. *X and Y are inverted lists. Upon first invocation, they correspond to the first two region types written in the original query. This algorithm produces a number of inverted lists that satisfy the chain-like query and is required to be kept.*

```
RecursiveMerge(X, Y)
  while (all lists are NONEMPTY)
    while (NONEMPTY(Y) and headof(Y) ≺ headof(X))
      advance Y;
    endwhile;
    while (NONEMPTY(Y) and headof(X) ▷ headof(Y))
      if (Y is the last term in the chain)
        output Y;
        advance Y;
        mark X;
      else
        if (RecursiveMerge(Y, next list in the chain))
          mark X;
        endif;
      endif;
```

```
    endwhile;
    if (X is marked)
      output X;
      signal := TRUE;
    else
      signal := FALSE;
    endif;
    if (all lists are NONEMPTY)
      while (NONEMPTY(X) and headof(X) ⊣ headof(Y))
        advance X;
      endwhile;
    endif;
    if (X is not the first list in the chain or not all lists are NONEMPTY)
      return(signal);
    endif;
  endwhile;
end;
```

The above algorithm follows a depth-first traversal in that it descends the chain first to find proper containment and traverses upward when it discovers whether it is possible to find an instance of the chain on this path. Every time a comparison is invoked, either an element of some list is processed (advanced) or a recursion occurs and a list is advanced before the return. Thus, the computation time needed to solve query A#..D#.C# is O(n) plus the time to calculate direct containment, where n is the upper bound for the number of elements in a list.

The alternative way is to decompose the query into a sequence of smaller, two element basic operations such that each component of the query is a basic operation first, like the ones listed in Section 3.3. The components are solved individually using algorithms described in the previous section. The results of each component are combined together in the end, using a multiway merge as above but with shorter lists, to eliminate partially qualified regions.

The example query combines both the *direct containment* (.) and the *general containment* (..) operators. It is quite logical to divide the query into two parts, namely, A..D and D.C. Let us process this in sequence. We will first resolve A..D using the basic algorithms described earlier, keeping both A*s* and D*s* as the intermediate result. Using the D*s* obtained in the first step, we will evaluate D.C in our second step. Ignoring the inefficiency

of the direct containment problem, we will obtain a set of C*s* from the second step, and possibly, a smaller set of D*s* will be obtained as the result of this evaluation, since some of the A..D*s* may not have a C as a child. If that is the case, the query engine will have to go back to the first step and reevaluate A..D using the A*s* obtained in the first step and the D*s* obtained in the second step to eliminate unqualified selections.

There are several choices in determining which subquery should be processed first. For example, a simple way is to process the subqueries in the order that they appear in the original query. This is simple to implement and easy to understand. If the probability of every query pattern is the same and the data is distributed evenly, then on average, this method is going to work as well as any others. However, in reality, most queries tend to repeat certain patterns and most collections of data are not evenly distributed. Therefore, other means of choosing the processing order are needed.

Generally, there are two criteria that could be taken into consideration for selecting the processing order of the subqueries.

- **Size of the participating lists** (i.e., *frequency* of the indexed element)
- **Selectivity** of the containment conditions

As the size of the inverted lists increase, so does the computation time to process them. More importantly, the likelihood of creating large intermediate results increases as well. Since the intermediate results are passed to the next subquery as input, having larger intermediate results at the start can have a ripple effect on the processing of all subsequent operations. Therefore, it is better to choose smaller lists to be processed first. A list's size is implicitly stored in the inverted list structure (e.g., by taking the difference between two adjacent list header addresses and dividing by the list element size) and can be retrieved without any difficulty.

However, it is possible to merge two large lists and produce small or even empty intermediate results. That depends on the *join selectivity* or how often those two structures appear within each other. This information can be collected and calculated at index building time, and stored with each inverted list.

In fact, the selection criteria mentioned above are very similar to the optimization issues in determining join order in conventional database systems. Therefore, the strategies mentioned above are similar to those often applied in conventional query optimization processes. However, unlike a normal join,

we do not carry forward both sets of qualified regions for execution of the subsequent subqueries (whereas tuples are joined together in the intermediate relations for a join). This distinction makes containment queries more similar to semijoins, where only partial information is passed on to the next relation to save transmission cost [Ull88]. Although, some of the techniques used in semijoins might also be effective for the chains, they will not be discussed further in this paper.

For simplicity, we will only incorporate the frequency of indexed terms into our algorithm below for the simple inverted lists.

Algorithm 4.1.2 Three attribute merge. *A', D', and C' are the results produced by functions RegionMerge() (page 27) and FindChildren() (page 29).*

```
ThreeWay(A,D,C)
  if (|A| ≥ |C|)
    RegionMerge(A, D);
    FindChildren(D', C);
    if (D' has changed);
      RegionMerge(A', D');
    endif;
  else
    FindChildren(D, C);
    RegionMerge(A, D');
    if (D' has changed)
      RegionMerge(D', C');
    endif;
  endif;
end;
```

There are possibly two phases in Algorithm 4.1.2 after the original query decomposes into subcomponents. In the first phase, the engine will solve each of the resulting subqueries in the order determined by list size. The second phase starts after one pass through the sequence of the queries. If any of the intermediate region sets (e.g., D') reduced in size, another pass through the sequence of queries in reverse order is necessary to eliminate selections that are no longer valid. In the second pass, the regions that are either requested by the user or that will be used by the next basic operation in the sequence are kept as the intermediate result. The total time required in the worst

case is the sum of computation time for executing RegionMerge() twice and FindChildren once, which turns out to be $O(4n + nl) \Rightarrow O(nl)$. However, if indirect containment is the only type of operation involved in the original query, then the computation time is $O(n)$. These times are independent of whether we execute A..D or D.C first.

Now, let us look at a more general query, such as

$$\textbf{for } R_0\ op_0\ R_1\ op_1 \ldots op_{m-1}\ R_m \textbf{ extract } R_{a0}, R_{a1}, \ldots, R_{ak} \tag{4.2}$$

where the R_is are the regions involved in the query, $k \leq m$ and op_i can be either a *direct containment* operator (.) or a *general containment* (..) operator. The region names listed after the keyword *extract* are the ones that are requested by the user.

As before, using the simple inverted list structure, we can break down a long chain query into a sequence of basic operations first in the form below:

$$R_0\ op_0\ R_1 \tag{4.3}$$
$$R_1\ op_1\ R_2 \tag{4.4}$$
$$\vdots \tag{4.5}$$
$$R_{m-1}\ op_{m-1}\ R_m \tag{4.6}$$

We will order the the subqueries by the sum of their two region lists in increasing order for the first pass. That way, in the absence of other information, the size of the intermediate results may be expected to be minimized. Of course, if the *selectivity* information is also available, it can be incorporated into the calculation to further optimize the query evaluation. For example, given a query

$$A..B.C.D..E..F \tag{4.7}$$

we can break it down into subqueries and order them as below according to the sum of participating region list size:

$$B.C \tag{4.8}$$
$$D..E \tag{4.9}$$
$$A..B \tag{4.10}$$
$$E..F \tag{4.11}$$
$$C.D \tag{4.12}$$

However, on the second pass, all subqueries must be sequenced according to their order in the original query from the point where the last subquery was located in the first pass (e.g., $C..D$). In this case, the last subquery happens to appear in the middle of the original query, then the second pass should be broken into two phases: One starts at $B.C$ and proceeds backward (towards $A..B$) until no more regions are marked to its left; another phase starts from $D..E$ and proceeds forward (towards $E..F$) until no more regions are marked to its right. Each phase would stop if the execution of the previous subquery did not change the answer set. By reordering the second pass this way, we have guaranteed that new information, if there is any, is always passed to the subquery in the sequence, and avoided the possibility of performing additional passes through the subqueries.

The algorithm is very similar to the ThreeWay function described above. All we need to do is to change the outer *if* statement to a small routine that orders the subqueries according to the sum of their input list sizes. The computation time would then be the following.

with direct containment:	$O(m \times nl)$
without direct containment:	$O(m \times n)$

where m is the number of operations.

Note that the recursive merge function discussed earlier (page 39) can also be used to solve any form of the chain. The preprocessing of direct containment relationships still needs to be done, because if it does not, one of the following situations would occur. First, if the processing of direct containment is done inside the recursive merge, then for every pair of possible parent-child, we have to run through all other region lists to check for invalid answers. That means loading every existing region list into the memory. Obviously, this would require too much I/O and/or place tremendous demand on main memory space. The second way is to perform the task after the recursive merge is done. However, if any region types that are marked to be kept as the answer occur after the last direct relationship in the query, the algorithm will have to reexecute the merge down the chain again and then go up the chain, after solving the direct relationship to eliminate dangling regions. Another advantage of resolving the direct containment first is that smaller lists are likely to be produced by the process, which saves computation time during the merge. The analytical run time of the RecursiveMerge as applied here is exactly the same as with query decomposition, because each recursive call corresponds to the execution of a decomposed subquery

in a forward sequence, and the back tracking corresponds to the backward execution of the subqueries. However, one drawback with the RecursiveMerge is that we cannot take advantage of the information on list size or selectivity. Thus larger intermediate lists are more likely to result, leading to increased running time.

4.1.2 Region lists partitioned by parent

With the partitioned inverted lists, we can, again, take advantage of the hierarchical information when processing the chain queries.

First of all, we can check the original query to determine whether every direct relationship described in it exists in the collection of text. For example, if there does not exist a parent-based partitioned inverted list for D.C then the remainder of Query 4.1 (A#..D#.C#) would not need to be executed. Secondly, the improved structure does not require a merge operation to find a child nor does it require filtering to find the parent, which makes direct containment queries a lot faster.

The steps involved here to solve long chain queries (e.g., Query 4.2) is exactly the same as with the simple inverted lists, except that the basic operations will be evaluated as described in Section 3.3.2. Both the recursive and the decomposition algorithm can be adopted here, except that in the case of recursive algorithms, no preprocessing of direct containment is needed. Therefore, the computational complexity in the worst case is $O(m \times n \log l)$ with or without direct containment operations.

4.1.3 Analysis and critique

Looking at the following summary, parent-based partitioned inverted lists outperform the competition when direct containment is involved, and they lose when it is not. However, it should be noted that if *even one* direct containment operation is involved in the original query, the partitioned inverted lists should typically perform better.

Interestingly, the computation time for both structures grows at the same rate from basic operations to chain queries. This results from the fact that the way to divide the original queries and the selection of processing order for the subqueries in both cases are the same.

Again, as a special case, there will be an additional gain for partitioned inverted lists if the first operation in the original query is direct containment

and the parent is not required by the user. Unfortunately, this advantage becomes less significant in practice as the length of the chain grows.

Chain Like Queries	Simple Inverted Lists	Partitioned Inverted Lists
With Direct Containment	O(mnl)	***O(mnlog(l))***
Without Direct Containment	***O(mn)***	O(mnlog(l))

m = number of operations in the original query

n = upper bound of unpartitioned inverted list size

l = total number of region types

Table 4.1: Worst case computation times for chains

4.2 Matching a tree

Besides the simple nesting relationship, lateral, or sibling, relationships are also present in structured text. These are best modelled by a tree. For example, in some email systems (e.g., pine), saved email message headers are arranged in different sequence depending on whether they are for incoming mail or outgoing mail. In particular, the order of sender and date field is reversed for those messages (see Figure 3.1). Therefore, to find all incoming email, we need only retrieve messages that have the sender field located before the date field.

Pekka Kilpeläinen has derived a query model relying on a single primitive *tree inclusion* that deals with sibling relationships [Kil92]. The idea is to model both the structure of the database and the query as trees, to find an embedding of the pattern into the database which respects the hierarchical

relationships between nodes of the pattern. The matching is done by deleting intermediate nodes and then connecting their children to their parent. This allows data independent queries to be issued, but in the worst case, the whole database structure must be searched in order to find all matches [KM93]. This structure also requires the text to be stored as tree structures where each node records where its children are. This approach requires extra storage and run-time space compared to single inverted lists for each region.

We will use the following notation in our discussion of tree-like queries [BCK+94].

$$P\#[A\#, B\#] \tag{4.13}$$

$$P\#[\hat{}A\#, \hat{}B\#] \tag{4.14}$$

The first query defines a particular text structure such that there is a type A region that is indirectly contained in a region of type P and is also followed by instances of region type B within that same P. All three regions are requested in the result.

Similarly, Query 4.14 requests all instances of text regions that satisfy the following conditions: there exists a type A region that is followed by a type B region, and both regions are children of (directly contained in) the same P type text region.

In the following subsections, we will examine how simple inverted lists and partitioned inverted lists by parent types deals with tree-like queries.

4.2.1 Simple inverted list

To solve the three attribute tree-like query, the engine will have to perform a list merge using three inverted lists, namely, the lists for the regions P, A, and B. We cannot merely decompose the query into basic operations, because it is not possible to resolve the "followed by within ancestor" relationship with only two attributes at a time. For example, query P#[A#,B#] can be decomposed into the following steps:

1. perform $P\#..A\#$
2. perform $P'\#..B\#$
3. perform $P''\#..A'\#$

4. resolve $A''\#$ followed by $B'\#$ when under the same $P''\#$ instance

where X' represents the intermediate result produced by the previous operation involving X. As in step 4, at least three lists are needed simultaneously to solve any tree-like query.

Algorithm 4.2.1 Tree like merge. *We will modify the two-way merge algorithm to add on the capability of computing the "followed by" clause.*

```
Followedby(A,B)
  while (NONEMPTY(B) and (headof(B) ≺ headof(A) or headof(B) ⊣ headof(A)))
  {B is before A, contains A, or is contained by A}
    advance B;
  endwhile;
end;

Tree(P,A,B)
  while (P, A, and B are NONEMPTY)
    while (NONEMPTY(A) and headof(A) ≺ headof(P))
      advance A;
    endwhile;
    if (NONEMPTY(A) and headof(P) ▷ headof(A))
      Followedby(A,B);
      if (headof(P) ▷ headof(B))
        while (headof(P) ▷ headof(B))
          output B;
          LastB := headof(B);
          advance B;
        endwhile;
        while (NONEMPTY(A) and (headof(A) ≺ LastB and headof(A) ⊣ LastB))
          output A;
          advance A;
        endwhile;
        mark P;
      endif;
      while (NONEMPTY(A) and headof(P) ▷ headof(A)) {clean up rest of invalid A}
        advance A;
      endwhile;
    endif;
```

```
    if (P is marked)
      output P;
    endif;
    while (NONEMPTY(P) and headof(P) ⊣ headof(A))
      advance P;
    endwhile;
  endwhile;
  discard all remaining elements;
end;
```

Note that we performed the Followedby operation after a valid A type descendent is found inside an instance of P. If a valid B instance is found, then we will output all valid B instances inside that same P. It is guaranteed to be correct because all instances of B following the first valid B will also follow the same A and thus satisfy the condition imposed as long as it is inside the same P. We can use the same trick to find all the valid A*s* under that P as long as it is followed by the very last B (represented by LastB). It can be seen from the algorithm that during the execution of any loop iterations, one list element is removed from the input list. Thus, this still yields a linear time algorithm in terms of the input size. Thus, the computation time is $O(n)$.

The algorithm used to solve P#[^ A#,^ B#] is very similar to the one used to solve P#[A#,B#]. In fact, we can view the direct containment relationship as a specialization of the general containment relationship. Thus, anything that can be used for general containment can also be used on direct containment. In this case, we will need to preprocess the direct containment before executing the merge, just as we did for the chains.

Algorithm 4.2.2 Tree for direct containment. *Let P', A', and B' be the resulting lists produced by FindChildren for regions of type P, A, and B respectively.*

```
DirectTree(P,A,B)
  FindChildren(P,A);
  FindChildren(P',B);
  Tree(P'', A', B');
end;
```

Since FindChildren requires $O(nl)$ time to run, and Tree takes $O(n)$ time, the computation complexity of this algorithm is $O(nl)$. With a more complicated query which involves more than three region types and/or more than

two levels of tree structure, the computation time will grow accordingly. For example, with a query such as

$$\textbf{for } R_0 \ [\ op_1\ T_1,\ op_2\ T_2, \ldots, op_m\ T_m] \textbf{ extract } R_{a0}, R_{a1}, \ldots, R_{ak} \qquad (4.15)$$

where the R_is are the region list involved in the query, a T_i can be either a tree or a region list, and op_i can be either a *direct containment* operator (^) or empty, which indicates that it is *general containment*. R_is listed after the keyword *extract* are the ones that are requested by the user.

A chain is a special class of tree where all regions involved are nested within each other, and there are no siblings. In other words, it is a one branch tree. With that in mind, we can construct an algorithm that combines the Tree algorithm used for three region tree and the RecursiveMerge for the chain. We will modify the RecursiveMerge algorithm to add on the capability of handling the "followed by" clause.

Algorithm 4.2.3 Recursive tree merge. *X, Y are pattern trees of the following form.*

```
pattern-tree = {
               list-of-region    regions;
               pattern-tree      child;
               pattern-tree      sibling;
               region            last;
               }
```

where *regions* is an inverted region list attached to each node in the pattern tree, *child* is the subtree rooted at the first child of the current as stated in the query, *sibling* is the subtree rooted at the first sibling of the current node (to the right) as stated in the query, and *last* stores the value of last valid region of the current type output.

For the sake of simplicity and ease of understanding, we will define a few utility routines for the recursive algorithm.

```
discardYbeforeX(X,Y)
  while (NONEMPTY(Y.regions) and headof(Y.regions) ≺ headof(X.regions))
    advance Y.regions;
  endwhile;
end;
```

The routine discardYbeforeX takes two pattern trees X and Y, and discards all regions in Y's region list that start ahead of the top element in the X's region list.

```
discardXbeforeY(X,Y)
  while (NONEMPTY(X.regions) and headof(X.regions) ⊣ headof(Y.regions))
    advance X.regions;
  endwhile;
end;
```

Similarly, this routine discards all regions at the front of the X inverted list that end before the top element of the Y list.

```
discardYinX(X,Y)
  while (NONEMPTY(Y.regions) and headof(X.regions) ▷ headof(Y.regions))
    advance Y.regions;
  endwhile;
end;
```

The above routine is invoked when we determined that all regions of Y that are contained by the top element of X are invalid. This situation would occur when the last instance of Y's sibling is found to be ahead of the current (top) element of Y's region list.

```
ProcessValidY(X,Y)
  output Y.regions;
  Y.last := headof(Y.regions);
  advance Y.regions;
  mark X.regions;
end;
```

This routine is used to output a valid instance of Y region and at the same time raise a flag for the X region that contains it.

```
ProcessX(X,Y)
  if (X.child = Y)  {if Y is X's first child}
    if (X.regions is marked)
      output X.regions;
      signal := TRUE;
```

```
    else
      signal := FALSE;
    endif;
    if (all lists are NONEMPTY)
      discardXbeforeY(X,Y);
    endif;
  endif;
  return(signal);
end;
```

Process X is invoked almost at the end of the recursive function to process the top level node (i.e., X) after all its siblings and children are processed (i.e., after all recursive calls are returned). Notice that we will only output the current top element of X's region list because that is the only one that can possibly be matched with the query pattern. We then advance the X list until the top element of X ends after the top element of Y's region list.

The following three algorithms, namely RecursiveTree, FindNextSibling, and OutputTree are the ones that perform the major work. RecursiveTree is the core of the algorithm that traverses the tree using depth-first traversal from left to right, trying to find one instance of the tree pattern in the database. FoundSibling is the routine that RecursiveTree calls to process siblings to the right of the current child node Y. Upon returning from the FindNextSibling routine, all subtrees to the right of the current child node Y should be processed (i.e., either a match is found or it is impossible to find a match under the current parent X). The routine OutputTree is used to output a subtree from right to left and bottom up. It is necessary because RecursiveTree cannot output any branch if there are any other branch to the right of it without first checking for a complete match. Therefore, after traversing a branches and discovering a partial match, RecursiveTree will have to backtrack and process the next branch without any output. Only after the right-most leaf node of the tree is reached and a complete match is found can we then start to output the regions.

```
OutputTree(X,Y)
  discardYbeforeX(X,Y);
  if (NONEMPTY(Y.regions) and headof(X.regions) ▷ headof(Y.regions))
    signal := TRUE;
    if (Y.sibling ≠ NULL)
```

```
      OutputTree(X, Y.sibling);
    endif;
    while (NONEMPTY(Y.regions) and headof(X.regions) ▷ headof(Y.regions))
      if (Y.child ≠ NULL)
        signal := OutputTree(Y, Y.child);
      endif;
      if ((signal) and ((Y.sibling = NULL) or
        (headof(Y.regions) ≺ headof(Y.sibling.last)
        and headof(Y.regions) ⊣ headof(Y.sibling.last))))
        ProcessValidY(X, Y);
      else    {simply discard the top element of Y}
        advance Y.regions;
      endif;
    endwhile;
  else
    signal := FALSE;
  endif;
  return(signal);
end;
```

The task of the *OutputTree* routine is to print all valid matches within the subtree rooted at X. Similar to the way chain works, we have to output the last node of the tree first and then proceed from bottom up for every branch. In addition, we must also output the elements from right to left to preserve the *followedby* property. Therefore, when arriving at any child node (Y) in the tree, we process its right sibling first, then its children, and finally the node itself. The value returned by this routine is used to indicate whether a match is found with the current instance of X.

```
FindNextSibling(X, Y)
  Followedby(Y.regions, Y.sibling.regions);
  if (RecursiveTree(X, Y.sibling))
    signal := TRUE;
    while (OUTPUTTING and headof(Y.regions) ≺ headof(Y.sibling.last)
          and headof(Y.regions) ⊣ headof(Y.sibling.last))
      if (Y.child ≠ NULL)
        signal := OutputTree(Y, Y.child);
      endif;
```

```
      if (signal)
        ProcessValidY(X,Y);
      else    {simply discard the top element of Y}
        advance Y.regions;
      endif;
    endwhile;
  else
    signal := FALSE;
  endif;
  if (OUTPUTTING)    {clean up rest of invalid Y.regions}
    discardYinX(X,Y);
  endif;
  return(signal);
end;
```

The above routine is used by the recursive algorithm both to find a match (i.e., will process the next appropriate node as soon as one valid partial pattern up to the current node is found) and to output each sibling after a match is found (i.e., must print out all possible paths that matches the pattern and no outputting is done). The boolean $OUTPUTTING$ is used to differentiate those two states. We first locate the first sibling to the right of the current child node Y, then call the RecursiveTree routine with the parent X and the right sibling $Y.sibling$ as the parameter. Upon returning from this call, three things can happen: we may have found a complete match, or we may have determined that no match can be found with the current path, otherwise, a partial match is found with this branch of the tree, but there are still parts of the query pattern that not been processed. In the first case, we will output every Y that appears ahead of the last instance of its right sibling. If we also need to match some children of Y, then we must traverse the subtree rooted at Y as well to output the children for each instance of Y. In the second case, we will simply return with the value of $FALSE$ to indicate that a new path needs to be sought. In the last case, a value of $TRUE$ will be returned instead when the function ends, so that the next region in the query pattern can be processed. In the first case, the function will always return $TRUE$ because there are at least one match with the current path.

RecursiveTree is the function first invoked to start this recursive algorithm. X is the root of the original query tree, and Y is X's left-most child

upon first invocation. After the some initialization, we start to compare the parent X and the child Y regions. There are three possible cases when comparing two regions.

1. X starts after Y
2. X contains Y: X starts before and ends after Y
3. X ends before Y

Therefore, we start with eliminating the first case by calling discardYbeforeX. In the second case, which is the one of interest, we may run into three possible scenarios:

1. Y is the right-most leaf node in the tree (last term in the query)
2. Y is any other leaf node
3. Y has children or siblings or both

Thus, for each $Y \lhd X$, we must check for every situation listed above. If Y is the right-most leaf node, we output Y (using ProcessValidY) and set the flag to indicate that a complete match is found. If Y is any other leaf node, then we set the flag to indicate a successful partial match and mark its parent X for output. In the third case, we process Y's children and sibling subtrees in that order. Note that if we couldn't find a match with the children, then we can skip processing Y's siblings and return to the caller. Upon returning from this function, either a match is found for subtree rooted at X or no match can be found and X is advanced to the next possible position.

```
RecursiveTree(X, Y)
  while (all lists are NONEMPTY)
    OUTPUTTING := FALSE;
    signal := FALSE;
    discardYbeforeX(X,Y);
    if (NONEMPTY(Y.regions) and headof(X.regions) ▷ headof(Y.regions))
      signal := TRUE;
      if (Y is the last term in the tree)
        while (NONEMPTY(Y.regions) and headof(X.regions) ▷ headof(Y.regions))
          ProcessValidY(X,Y);
          OUTPUTTING := TRUE;
```

```
        endwhile;
      else
        if (Y.child ≠ NULL)
          signal := RecursiveTree(Y, Y.child))
        endif;
        if (signal and Y.sibling ≠ NULL)
          signal := FindNextSibling(X,Y);
        endif;
        if (signal)
          mark X.regions;
        endif;
      endif;
    endif;
    signal:=ProcessX(X,Y);
    if (X is not the first term in the original query or not all lists are NONEMPTY
        or X.child ≠ Y)
      return(signal);
    endif;
  endwhile;
end;
```

Of course, whenever there are direct containment relationships presented in the query, we have to process them first as we did for the recursive algorithm for chains.

Since each node in the tree will be traversed twice, once to find a match and once for output, for a m-node tree, the computation time will be $O(2mn) \Rightarrow O(mn)$ for trees without any direct containment relationship, and $O(mn + fnl)$ *or* $O(n(m + fl))$ where f is the number of direct relationships.

4.2.2 Region lists partitioned by parent production

As with the chains, the parent-based partitioned inverted lists provide an advantage over simple inverted lists when direct containment is involved. By prechecking the satisfiability of all direct containment relationships, by determining whether a particular partitioned inverted lists exists or not, the algorithm can terminate in constant time if the answer set is empty. Based on the same algorithms (e.g., RecursiveTree), the absence of direct containment

Tree Like Queries	Simple Inverted Lists	Partitioned Inverted Lists
With Direct Containment	O(mnl)	***O(mnlog(l))***
Without Direct Containment	***O(mn)***	O(mnlog(l))

m = number of operations in the original query

n = upper bound of unpartitioned inverted list size

l = total number of region types

Table 4.2: Worst case computation times for trees

means that the total computation time would be $O(mn \log l)$ with partitioned inverted lists.

4.2.3 Analysis and critique

In the previous two sections, we analyzed the processing behaviour of both simple and parent-based partitioned inverted lists with more complicated query patterns, namely tree-like queries. Table 4.2 summarizes the computation time with and without direct containment relationships.

It is interesting to note that the chart indicates the same execution time as for chain-like queries. Chains are a special kind of tree, but the additional complexity added on by the siblings only increases the programming complexity by a constant factor.

4.3 Application examples

We have thus far discussed the simple and partitioned inverted lists with corresponding algorithms and analyzed their performance in terms of the

input list size. In this section, we illustrate the algorithms with a few practical examples taken from the *Oxford Advanced Learner's Dictionary* [Cow89] to demonstrate the performance difference in those two structures for some live data. There are 74 distinct region types in OALD, and all regions are enclosed between an opening tag and a closing tag (e.g., <ent> and </ent> marks the boundary of an entry). In total, there are 632,643 tags in the online OALD (see Appendix).

Let us start with a few simple queries. For example, in the OALD, a cross reference takes several forms.

TOP[1] indicates a cross reference to the first entry for TOP, and is encoded in the source as <RFW>top<hn>1</hn></RFW>

MIND-BOGGLING(MIND[1]) references to the compound MIND-BOGGLING under the first entry for MIND. That is encoded as <RFW>mind-boggling <hw>mind<hn>1</hn></hw></RFW>

If we wish to find all cross references directly to words that have several entries in the OALD, we will issue the query *RFW.hn* to avoid picking up those references to compound forms. On the other hand, if we are interested in all cross reference to words with multiple entries, regardless of the reference form, then we will issue *RFW..hn* instead.

The columns in Table 4.3 under the heading of *input list size* are the size of input lists to the merge algorithm. Note that only the lists relevant to the processing of query are shown in the cell. The columns under *output list size* are size of lists produced by the algorithm. The number in **bold** is the size of final result. The fourth major column in Table 4.3a displays information about the total number of lists and the total number of regions involved in the filtering process other than the lists involved in the current direct relationship subquery. Since for direct containment, the algorithm will filter through all other region types, the number of filter region types are in fact the number of possible regions in the text subtracting two. That region count is obtained by subtracting the original size of the inverted lists in the subquery (i.e., *RFW* and *hn*) from the total number of regions in the text (i.e., 632,643). On the other hand the corresponding two columns in Table 4.3b show the number of partitioned inverted lists that need to be manipulated to form virtual lists during merging.

With the simple inverted lists, solving *RFW..hn#* (general containment) requires a straight two-way merge, and the cost is proportional to the sum

Subqueries	Input list size		Output list size		Filter list size		Cost
	RFW	*hn*	*RFW'*	*hn'*	# of types	# of regions	
RFW.hn#	*9,720*	*22,823*		***2,574***	*72*	*600,100*	*1,052,691*
RFW..hn#	*9,720*	*22,823*		***2,917***			*32,543*
RFW#.hn	*9,720*	*22,823*	***2,574***		*72*	*600,100*	*1,052,691*
RFW#..hn	*9,720*	*22,823*	***2,917***				*32,543*

a) Input and output list sizes using simple inverted list structure

Subqueries	Input list size		Output list size		# of partitions involved		Cost
	RFW	*hn*	*RFW'*	*hn'*	ancestor	descendent	
RFW.hn#		*2,574*		***2,574***			*0*
RFW..hn#	*9,720*	*22,823*		***2,917***	*11*	*3*	*56,792*
RFW#.hn	*9,720*	*2,574*	***2,574***		*11*	*1*	*36,200*
RFW#..hn	*9,720*	*22,823*	***2,917***		*11*	*3*	*69,800*

b) Input and output list sizes using partitioned inverted list structure

Table 4.3: Performance comparison for basic containment query

of both input lists, which is $9,720 + 22,823$. On the other hand, solving *RFW.hn* (direct containment) needs a filtering process after the two-merge. Since we are comparing the lists RFW' and hn' against every other region list in the system, in the worst case, every element of those two lists is considered 72 times (i.e., once for every other list used for filtering). For example, if we load the filtering lists in the order listed in Appendix C, then we will first use list *ALD*, which has 174 elements, and none of the elements in *hn* will be eliminated. Next, we filter through *ALT*, and again, both *RFW* and *hn* lists are not reduced. In fact, nothing will be eliminated until we filter with the list *hw*. Hence, in the worst case, it is possible that nothing with be filtered out until we come to the last list. Of course, the algorithm will not compare the elements of any remaining lists if one of the lists is exhausted. For example, the list *BTY* only has one instance, and the chances are very high that not all elements of RFW' and hn' will need to compared. However, if this *BTY* region appears after all *RFW* and *hn* regions, then all comparisons still need to be carried out. Therefore, the worst case cost of *RFW.hn#* is proportional to $(9,720 + 22,823) + (2,917 + 2,917) \times 72 + 600,100$.

With the simple inverted lists, the cost of finding child or parent is the same, and the cost of finding ancestor or descendent is the same, but it is different for partitioned lists. In Table 4.3b, we know that solving *RFW.hn#* (child) does not require any comparisons, but finding *RFW#.hn* (parent) does require merging. Since, there are multiple sublists for *RFW*, we need to create a virtual list, with a cost proportional to the logarithm of the number of sublists, in order to facilitate the merge. Thus, *RFW#.hn* has the cost of $9,720 \times \log 11 + 2,574$ comparisons. This is approximately one third of the cost as compared to the simple inverted lists. On the other hand, Finding *RFW#..hn* (ancestor) requires $9,720 \times \log 11 + 22,823 \times \log 3$ comparisons which is a little more than twice the cost when compared to simple inverted lists. Finding a solution for *RFW..hn#* is similar, but by taking advantage of knowing that *RFW.hn#* is in the answer set, we can reduce the number of comparisons to $9,720 \times \log 11 + (22,823 - 2,574) \times \log 2 + 2,917$. $2,917$ is added to the cost because we need to eventually merge the list *RFW.hn#* into the calculated answer set.

Table 4.4 shows the list sizes for a chain-like query *ent#..lsen.GEO*, asking which entries contain at least one subsense that has a specific meaning to some geographic region. In the table, the original query has been broken down into components, and the rows are arranged in the order of execution. Unlike Table 4.3, the number under the heading *input list size* can be either

Subqueries	Input list size			Output list size			Filter list size		Costs
	ent	*lsen*	*GEO*	*ent'*	*lsen'*	*GEO'*	# of types	# of regions	
ent#..lsen#	*22,283*	*12,635*		*3,952*	*12,635*				*34,918*
lsen'#.GEO#		*12,635*	*4,354*		*207*	*208*	*72*	*615,654*	*665,547*
ent'#..lsen''	*3,952*	*207*		***162***					*4,159*
Total Cost									*704,624*

a) Input and output lists sizes using simple inverted list structure

Subqueries	Input list size			Output list size			# of partitions involved		Costs
	ent	*lsen*	*GEO*	*ent'*	*lsen'*	*GEO'*	ancestor	descendent	
ent#..lsen#	*22,283*	*12,635*		*3,952*	*12,635*		*1*	*8*	*60,188*
lsen'#.GEO#		*12,635*	*208*		*207*	*208*	*1*	*1*	*12,843*
ent'#..lsen''	*3,952*	*207*		***162***			*1*	*1*	*4,159*
Total Cost									*77,190*

b) Input and output lists sizes using partitioned inverted list structure

Table 4.4: Performance comparison for simple chain query

the original size of the lists or the intermediate list sizes produced from previous operations.

Notice the input sizes for GEO using the two structures are very different for the subquery $lsen'.GEO$. This is because we are using one partitioned list with parent-based inverted lists instead of the whole list as with simple inverted lists. The total number of comparisons is the sum of comparisons for each step. For the simple inverted lists, $ent\#..lsen\#$'s cost is the sum of two input lists, while $lsen'\#.GEO\#$ have to go through the filtering process, again. Two numbers that are not shown in the table are the output list size of $lsen'$ and GEO after performing a two-way merge. They are 225 and 232 respectively, which are important in calculating the input list size for the filtering routine. Thus, the cost of $lsen'\#.GEO\#$ is calculated as: $(12,635 + 4,354) + (225 + 232) \times 72 + 615,654$. The last step, $(ent'\#..lsen'')$, again, has the number of comparisons equal to the sum of its input lists.

With the partitioned inverted lists, notice that except for the first subquery $(ent\#..lsen\#)$, the rest of the operations do not need any virtual list manipulation. This is because the algorithm uses the output list of the pre-

vious operation as the input list, and those outputs are already sequenced. As a result, the algorithm is just as efficient as simple inverted list when performing general containment later in the query (i.e., $ent'\#..lsen''$). We can see that using the partitioned inverted lists drastically improves the performance of this chain-like query, from $704,624$ comparisons down to $77,190$ comparisons. This shows the strength of parent-based partition for direct containment and how there may be little or no degradation when dealing with general containment.

With our algorithms, postings lists for words, can be treated as region lists when processing queries. For example, if we want to find all derivatives that are intransitive verbs, a query in the following form may be issued: $drivgp\#.PAT.\text{`}\&I\text{'}$. This query asks the engine to look for a derivative group ($drivgp$) that directly contains a verb pattern (PAT), which in turn directly contains an intransitive verb symbol ('$\&I$'). This query would officially processed be if we have a parent-based partitioned inverted list structure. However, if we are using a simple inverted list index structure, then following Consens and Milo [CM94] we could change the query to $drivgp\#.PAT..\text{`}\&I\text{'}$ since we might know that any verb symbol must be directly contained in the verb pattern tags (PAT), and the processing of general containment is much more efficient than direct containment with this structure. In this case, the cost of the query for simple inverted list is the sum of input lists PAT and '$\&I$' $18,585 + 2,840 = 21,425$, and plus the cost of merging $drivgp$ with PAT', $12,458 + 1,748 = 14,206$, and finally, the cost of filtering $drivgp'$ and PAT'', $(371 + 533) \times 72 + 601677 = 666,765$. Therefore, the total cost is $702,396$, and the final answer set includes 164 instances of $drivgp$ regions.

On the other hand, with partitioned inverted lists, we can take two partitioned lists $drivgp.PAT\#$ and $PAT.\text{`}\&I\text{'}\#$, and merge them first ($1,748 + 2,840 = 4,588$). Then, use the resulting PAT' list and merge with drivgp. The number of comparisons is as follows: $208 + 12,458 = 12,666$. Note that, since $drivgp$ only has one possible parent, no virtual tree is needed. The total cost is $4,588 + 12,666 = 17,254$ comparisons, which is less than 3% of the cost for simple inverted list.

Finally, if we are looking for the roots form for the above derivatives, we would modify the query to $ent[HWD\#, drivgp[PAT[\text{`}\&I\text{'}]]]$ where HWD is the tag that encloses the headword (or the root) of an entry. The original sizes of the ent list and HWD list are $22,283$ and $22,289$ respectively. Both lists have only one partition (ent has no parent because it is the outermost structural unit). Since both simple and partitioned inverted list use the

same algorithm for tree-like queries, except the need to preprocess direct containment with simple list, we will just show the result for the partitioned inverted lists. According to our RecursiveTree algorithm, the cost of the query is the sum of all lists involved, adding the cost of maintaining any virtual tree which happens to be zero in this case. For this particular query the cost is $22,283 + 22,289 + 12,458 + 1,748 + 2,840 = 61,618$. The final result contains 163 HWD regions.

Chapter 5

Overlapping Lists

In the previous chapters, we have assumed that no overlapping regions are allowed. In this chapter, we will show why we have chosen not to relax this restriction.

5.1 Overlap of regions of distinct types

Allowing overlapping of regions of distinct types would enhance our inverted lists model by making it able to process non-hierarchically structured text. For example, a region list for *pages* will most likely overlap with another region list for *paragraphs* in a publication document. Thus, allowing for overlaps is a useful extension.

In fact, pattern matching against non-hierarchical texts can be handled by some of the existing systems, such as PAT. Similarly, the simple inverted lists structure and algorithms, general or direct containment, described in this thesis will also not be effected by the addition of this characteristics.

However, when it comes to parent-based partitioned inverted lists, the characteristics of one parent per child is lost. For example, in Figure 5.1, according to our definition of parent, both U_1 and V_2 would qualify as the parent of Y_4. We can resolve this potential problem if we store duplicates of a single region in different sublists of the inverted lists (e.g., store Y_4 in lists U.Y and V.Y). Unfortunately, this increases the worst case list size from $O(n)$ to $O(nl)$. This will require more main memory to hold each list in total, or we will need more I/O to process a single list. Furthermore, the computation time for solving general containment relationships and finding a

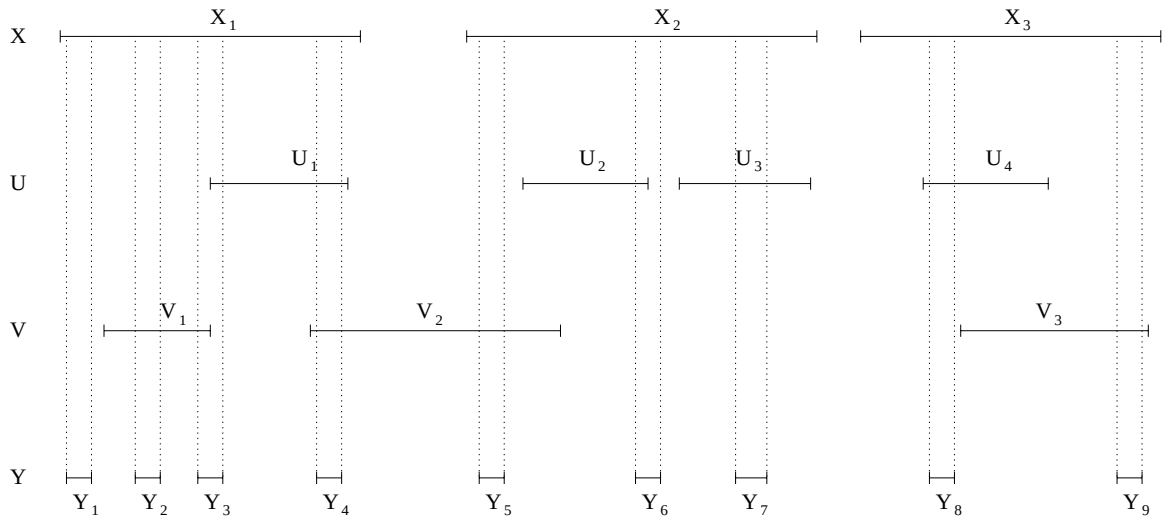


Figure 5.1: Overlapping of distinct region types

parent with parent-based inverted lists change from $O(n \log l)$ to $O(nl \log l)$. This degradation in performance extends to chain-like and tree-like queries as well.

5.2 Overlap of regions of one type

We have seen that allowing overlapping for distinct region lists has caused problems for partitioned inverted lists. Now, let us discuss what impact results from overlapping within a single inverted list.

An overlapping inverted list, as is Figure 5.2, is an inverted list where the regions marked by each element can overlap with each other without being completely contained by one another. In other words, the only ordering condition in an overlapping list is that each element region starts before the next element starts and also ends before the next element ends. For example, in an inverted list $X = \{x_0, x_1, \ldots, x_{m-1}, x_m\}$, X is an overlapping inverted list if and only if $\{\forall i < m \ (x_i \prec x_{i+1}) \wedge (x_i \dashv x_{i+1})\}$. The observation that such overlapping regions are not necessarily problematic is one of the insights of GC-lists (General Concordance Lists) [CCB95] over PAT [ST93]. Alternatively, Dao et al. [DSDT96] introduced SC-lists (Simple Concordance Lists) which extend GC-lists to allow self-nesting within a single type. We examine overlapping with GC-lists only.

We start with the simple indirect containment query introduced in Chapter 3, A#..D#. This was straightforward to implement with the non-overlapping simple inverted lists. All that was required was a two-way region merge (page 27). However, processing overlapping lists using the same algorithm would yield incorrect results because we cannot simply output and advance the descendent list: any descendent might also be a valid descendent for one or more of the next elements in the ancestor list. If we ignore this, we might discard some valid ancestor regions. The following algorithm works around the problem using a trick similar to that used for "followed by" and still keeps the algorithm linearly bounded. The key here is to realize that if one ancestor region is to contain any descendent regions that are also contained by another ancestor region ahead of it in the list, it must contain the very last descendent of that other ancestor region. Therefore, if we record what is the last descendent found by each ancestor region, we can make a comparison between the last descendent and the next ancestor to make sure no ancestor regions get thrown out mistakenly.

Algorithm 5.2.1 Overlapped list merge. *Lists X, Y are inverted lists allowing overlaps. This algorithm will produce two lists* $X' \subseteq X$ *and* $Y' \subseteq Y$ *having the same property such that all elements in* Y' *are contained in one or more elements of* X' *and all elements in* X' *contain one or more elements of* Y'.

OverlapMerge(X, Y)
  while *(X, Y are NONEMPTY)*
    while *(Y is NONEMPTY* **and** *headof(X)* $\rhd$ *headof(Y))*
      output Y;
      lastY := headof(Y);
      advance Y;
    endwhile;
    while *(X is NONEMPTY* **and** *headof(X)* $\rhd$ *lastY* **and** *headof(X)* $\not\rhd$ *headof(Y))*
      output X;
      advance X;
    endwhile;
    while *(X is NONEMPTY* **and** *headof(X)* $\lhd$ *headof(Y))*
      advance X;
    endwhile;
    if *(head(X)* $\not\rhd$ *headof(Y))*

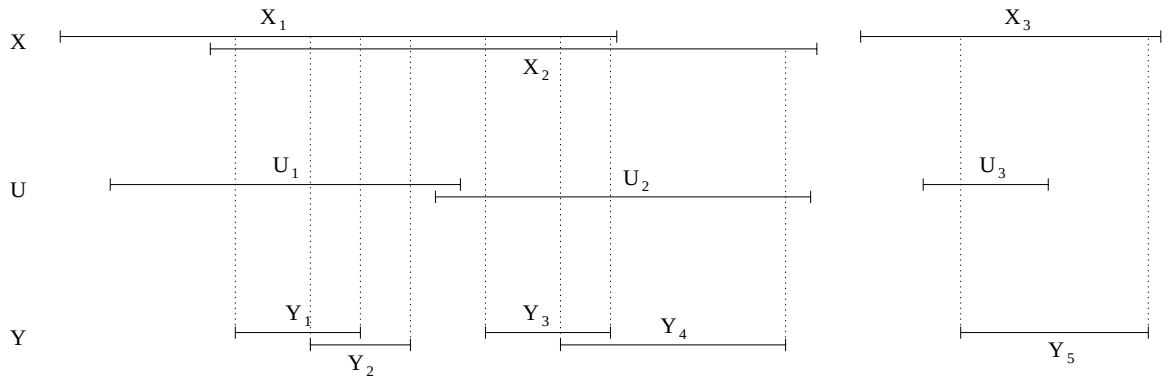


Figure 5.2: Example of filtering with overlapping region lists

```
      advance Y;
    endif;
    while (Y is NONEMPTY and headof(Y) ≺ headof(X))
      advance Y;
    endwhile;
  endwhile;
  discard all remaining elements;
end;
```

In the above algorithm, we record the most recent valid descendent regions ($LastY$), and in the second inner while loop, check to see whether the next ancestor region (X) that would otherwise be discarded also contains $LastY$. If so, we output that X region and then advance, otherwise, we simply advance the X list. This way we can maintain the computation time to be the same as for non-overlapping inverted lists at $O(n)$.

However, direct containment is not as simple. For example, to solve the basic operation P.C# for non-overlapping inverted lists, we invoked a filtering process to eliminate regions that satisfy general containment, but not direct containment. In Algorithm 3.3.2 in Chapter 3, we first invoked the general containment routine to find the answer to P..C and then filtered the intermediate results against all other region lists. Consider now the overlapping regions. Since we have to process at least three region lists at the same time during the filtering process, the problem depicted in Figure 5.2 might occur. Let us use the algorithm for the simple non-overlapping inverted lists to filter out all region lists such as U that might come between

X and Y. Assume that the X and the Y lists are output of the OverlapMerge routine such that all Xs contain some Y, and all Ys are contained by some X. We can see that $X_1 \rhd U_1 \rhd Y_1$ in the figure, in which case, the algorithm would toss out Y_1 by advancing the Y list. However, we also notice that $X_2 \rhd Y_1$ and $X_2 \not\rhd U_1$. Thus Y_1 is a valid child of X_2 and should be kept as the answer. The region Y_2 would similarly get discarded by the normal Filter routine. Therefore, we cannot simply discard any descendent regions if a double containment (i.e., $X_i \rhd U_i \rhd Y_i$) is found. Instead, we must append that Y_i to another list T. Once X_i has processed all its descendents, (i.e., when X_1 encounters Y_4), we must process X_{i+1} starting with elements of T, then the X list. If T_i happens to be a child of X_{i+1}, then it will be added to the output list and removed from the T list. Otherwise, either of two things might occur,

- if $T_i \prec X_{i+1}$, then remove T_i from the list, since it cannot be in any other X region,
- otherwise, leave T_i intact, and process T_{i+1}.

This process continues, until the whole T list is processed. After that we start again with the Y list. The way that all U lists are handled is similar to the X list (i.e., advance U if and only if $T_i \prec U_i$), except that it never outputs anything.

Thus, in the worst case, all elements of Y must be compared against all elements of the rest of the inverted lists (i.e., X and all U lists). That would drastically degrade the performance of the algorithm from $O(nl)$ to $O(n^3l)$.

With the improvements that we made to the inverted list structure by partitioning it based on parents, we again need to allow for repeated regions, and thus not a partitioning, when overlaps are permitted. Again, we don't have any problem finding the child of any region type. The time would still be constant. However, once we want to find the *parent* of some region type (e.g., P#.C), we run into the same problem as we did with the simple inverted lists. The problem here is manifested in a different form though. In the non-overlapping lists, the engine will first retrieve P.C# and create a virtual list for all lists of type P. A two-way merge is then performed on those two lists, because for each child C_i, there must be one and only one parent region P_i. With the introduction of overlapping, this is no longer the case: any C_i can be contained in more than one region of type P. Thus to determine the valid parent regions, we have to go through the same filtering

process as with the simple inverted list. Obviously, the computation time would increase accordingly to $O(n^3 \log l)$. Although, it is still better than the simple inverted lists, the difference is incomparable to the magnitude of increase in computation time over dealing with non-overlapping regions..

Patterns with chains would only make the problem worse. In fact, under most circumstances, a simple chain that only asks for children of certain regions, would still be inefficient even for our parent-based partitioned inverted lists. For example, to solve a chain of the form $A..P.C\#$, we can either solve $A..P$ first, or solve $P.C$ first. If we solve $A..P$ first, then we have to merge P', the intermediate results with list C list. That would require filtering. If we process $P.C$ first, then we have to keep P' in order to merge with A. That means finding the parent region, and again, filtering would be required. Furthermore, a query containing only direct containment relationship would still cause inefficiency because of the inability to avoid filtering. Consider Query P.C.D and the data as depicted in Figure 5.3. Note that P_1 contains both C_1 and C_2, and in turn both C_1 and C_2 contains D_1. In addition, D_1 has a parent of type C (i.e., C_1), and C_2 has a parent of type P in P_1. However, U_1 comes between C_1 and P_1 and U_2 comes between D and C_2. If we simply take the parent-based partitioned list $P.C$ and merge it with $C.D$, D_1 would be included in the answer set. However, we can see from the figure that it should not be kept. The problem is that each D could be contained by more than one C. Since D_1 has a parent of type C and C_2 is an ancestor of D_1, we have mistaken C_2 to be a valid parent of D_1. Therefore, filtering is required at this stage to eliminate those false matches. Because chain-like queries are a special class of tree-like queries, we can see that having overlapping lists with tree patterns would only create more difficulty.

In this section, we analyzed the impact of adding the overlapping property to the inverted lists of same type would have on all algorithms and structures. We showed that overlapping, except in limited applications, would significantly degrade the performance of our algorithms for both simple and parent-based inverted lists whenever direct containment is present in a query. The computation time turned from linear ($O(nl)$ for simple and $O(nl \log l)$ for parent-based inverted lists) to cubic ($O(n^3 l)$ for simple and $O(n^3 l^3 \log l)$ for partitioned inverted lists).

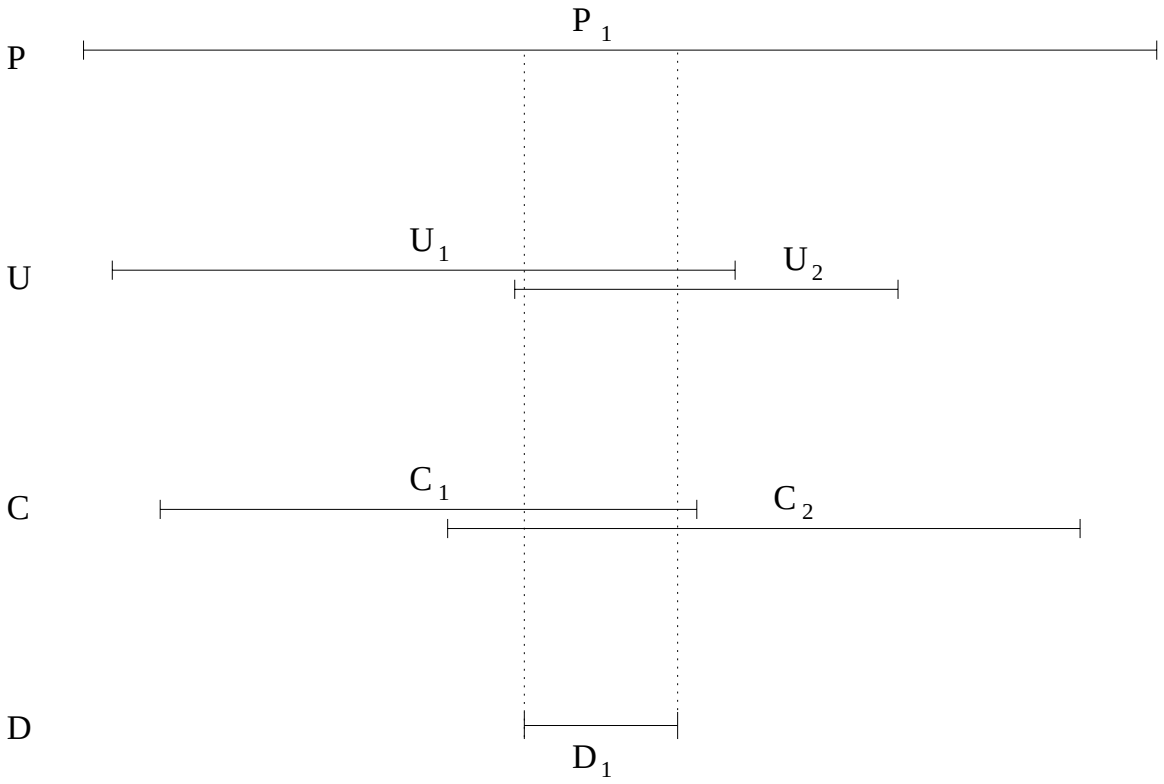


Figure 5.3: Solving chain $P.C.D\#$ with overlapping regions

Chapter 6

Conclusions and extensions

Exploitation of the text structure certainly adds another dimension to the processing of text beyond string matching. Integrating conventional postings with regions allows structural search to be performed alongside word and phrase matching. Several forms of text structure query have been examined. This provides the basis for a more powerful search engine that makes text retrieval easier for the users.

In this thesis, we described and analyzed the algorithms that can be used to handle general and direct containment queries for traditional simple region lists, such as those used in the PAT system, and parent-based partitioned inverted lists. We started by comparing the algorithms and their performance for basic operations that involve only two region types. We then extended those basic algorithms and built some complex (e.g., recursive merges) algorithms in order to handle chain-like queries where three or more regions types are linked in a query with strictly nested relationships (i.e., general and direct containment only). The performance of the algorithms for both forms of inverted structure are compared and analyzed. After that, a more complicated form of query, tree-like query where hierarchical sibling relationships such as $Q[A, B[\hat{}C, D], E]$ are allowed, is used to compare the performance of the two inverted list structures. Again, the performance is analyzed.

With all three forms of query, we have found that the simple inverted lists are more suited for general containment relationships, where the level of nesting is not important. However, as soon as direct containment relationships are present in the queries, parent-based partitioned inverted lists came out on top. With a relatively small amount of additional information in the index, better algorithms can be found for some specific types of query.

Finally, we discussed the properties of overlapping inverted lists, and how it would impact on containment operations. Overlapping inverted lists were first incorporated in the MultiText project [CCB95] to extend PAT regions and make the query engine more flexible. Like PAT, the MultiText system chose to ignore the direct containment relationship, and thus it did not encounter the kind of problems that we have described here. The problem is that when dealing with direct containment queries, allowing overlapping in our inverted lists structures, simple or partitioned, would turn our linear time algorithms into cubic time algorithms in terms of the input size.

We have used worst case analysis instead of average case in all of our performance analyses. However, in most structure text documents, there are rarely more than four or five possible parents for each region type. This was illustrated by examining the actual structures in the OALD, as summarized in Appendix C. Thus, the computation time of the partitioned inverted lists will more typically be close to the simple inverted lists for general containment query, and improve more significantly for direct containment queries.

In this thesis, we have studied the effects of the inverted lists for containment queries. The effects of partitioning the inverted lists when queries include non-containment operators such as those for proximity matching can also be investigated further.

This thesis has dealt mainly with theoretical algorithm design, and more practical experiments with real data should be conducted to validate the model. Many factors such as memory size and cache replacement policy can effect the performance of our algorithms. Care has to be taken to select a suitable environment for efficient performance.

It can be seen that the performance of either indexes are not very efficient when dealing with complex structural queries such as the tree. Further modification to the parent-based partitioned inverted lists might be possible to deal with this type of query. However, we must be careful not to add too much overhead for processing other type of queries when optimizing for this one.

We have mentioned that *join selectivity* and list size can be used to determining the join order when processing a chain-like query. Whether more statistical information can be utilized for optimization purposes is a topic worth further studying. Based on the similarity between the order to process the subqueries and the join order problem, especially semijoin order, it is worth further investigation whether other improvements based on join algorithms can be incorporated into our model.

Finally, we have ignored the problem of creating and maintaining indexes. Before our algorithms can be incorporated into production systems, these aspects must also be addressed.

Appendix A

Basic definitions and notation

A.1 Definitions

ancestor the outer region involved in a containment relationship

child the inner region involved in a direct containment relationship

containment if the start of A occurs before or at the start of B and the end of A occurs after or at the end of B, then A contains B

descendent the inner region involved in a containment relationship

direct containment when a region A contains a distinct region B, and there does not exist a distinct region C that contains B and is contained in A, we say that region A directly contains region B

parent the outer region involved in a direct containment relationship

partitioned inverted list a sequence of simple inverted lists, all for the same region type and each list containing pointers to those regions having *parents* of one specific type

region a contiguous piece of text of arbitrary length

siblings two distinct regions with the same ancestor and does not contain each other

simple inverted list a contiguous sequence of pointers to regions of one specific type, kept in order of position in the text

A.2 Notations

$a \rhd b$ means a *contains* b: $\{a.s \leq b.s$ and $a.e \geq b.e\}$

$a \lhd b$ means a *contained by* b: $\{a.s \geq b.s$ and $a.e \leq b.e\}$

$a \not\rhd b$ means a does *not contain* b: $\{a.s > b.s$ or $a.e < b.e\}$

$a \not\lhd b$ means a is *not contained by* b: $\{a.s < b.s$ or $a.e > b.e\}$

$a \prec b$ means a *starts before* b: $\{a.s < b.s\}$

$a \dashv b$ means a *ends before* b: $\{a.e < b.e\}$

A.3 Basic Operations

$A..D\#$ means to return a list of type D regions such that each element of that list is contained by some type A region

$A\#..D$ means to return a list of type A regions such that each element of that list contains some type D region

$A\#..D\#$ means to return a list of type A regions and a list of type D regions such that each element of the type A list contains some element of the type D list and each element of the type D list is contained by some element of the type A list

$P.C\#$ means to return a list of type C regions such that each element of that list is directly contained by some type P region

$P\#.C$ means to return a list of type P regions such that each element of that list directly contains some type C region

$P\#.C\#$ means to return a list of type P regions and a list of type C regions such that each element of the type P list directly contains some element of the type C list and each element of the type C list is directly contained by some element of the type P list

Appendix B

Table of algorithms and complexities

Here, we list the algorithms that was presented in this thesis. For each algorithm there is a brief description of functionality, input, expected output, and the computation complexity. The page numbers refer to the place where the complete algorithm is discussed in detail.

B.1 Two-way merge algorithm (page 12)

function: Simple merging of two lists in main memory

input: Two lists of maximum length n sequenced in the same order

output: One list of maximum length $2n$ with the same ordering as the input lists

computation complexity: $O(n)$

B.2 Multiway merge algorithm (page 13)

function: Merging of more than two lists into a single ordered list

input: P ordered lists of length n

output: One list of maximum length nP with the same ordering as the input lists

computation complexity: $O(nP)$

B.3 Selection tree algorithm (page 15)

function: A faster way of performing multiway merging by comparing pairs of list instead of comparing one list against every other lists

input: P ordered lists of length n

output: One list of maximum length nP with the same ordering as the input lists

computation complexity: $O(n \log P)$

B.4 Intersection algorithm (page 16)

function: Find the intersection of two lists ordered by the same criterion

input: Two ordered lists of length n

output: One list of maximum length nP with the same ordering as the input lists

computation complexity: $O(n)$

B.5 Generalized two-way intersection algorithm for text regions (page 27)

function: For any two ordered inverted lists of regions X and Y, the algorithm will produce two similarly ordered X' and Y' such that all elements in X' contains some element of Y' and all elements in Y' is contained by some element of X'

input: Two ordered region lists of length n

output: Two ordered region lists of length at most n

computation complexity: $O(n)$

B.6 Filtering algorithm (page 29)

function: It takes the results of the *Generalized two-way algorithm intersection for text regions* of compare them against every other region list in the system to satisfy the direct containment constraint.

input: Two ordered region lists of length at most n

output: Two ordered region lists of length at most n

computation complexity: $O(nl)$ where l is the number of distinct region types in the system

B.7 Finding Parent and Child algorithm (page 33)

function: Given all partitioned inverted lists for regions of type P and C. This algorithm finds all P regions that directly contains C. Both P and C are kept in the result

input: All partitioned region lists with total length at most $2n$

output: Two ordered region lists of length at most n each

computation complexity: $O(n \log(l))$

B.8 Find descendent algorithm (page 34)

function: Given all partitioned inverted lists for regions of type A and D. This algorithm finds all D regions that is contained A

input: All partitioned region lists with total length at most $2n$

output: One ordered region lists of length at most n

computation complexity: $O(n \log(l))$

B.9 Find ancestor and descendent algorithm (page 35)

function: Given all partitioned inverted lists for regions of type A and D. This algorithm finds all A regions that contains D. Both A and D are kept in the result

input: All partitioned region lists with total length at most $2n$

output: Two ordered region lists of length at most n each

computation complexity: $O(n \log(l))$

B.10 Recursive multiway merge algorithm (page 39)

function: Recursively perform the *Generalized two-way algorithm intersection for text regions* down a chain-like query. If there are direct containment relationships involved in the query, then they have to be resolved before performing this algorithm

input: Two inverted lists for the first two attribute in the query and a data structure that record what is the next attribute in the chain

output: A number of inverted lists of length at most n corresponding to the query

computation complexity: $O(nl)$ where l is the number of participating lists in the query

B.11 Three attribute merge algorithm (page 42)

function: Execute a three attribute chain-like query by decomposing them into two-attribute basic operations first and then execute each one in decreasing order of the sum of participating lists size.

input: Three inverted lists of length n

output: Three inverted lists of length at most n

computation complexity: $O(nl)$ with direct containment and $O(n)$ without direct containment

B.12 Tree like merge algorithm (page 48)

function: Added the capability of computing the "followed by" clause on top of the *Generalized two-way intersection algorithm for text regions* to solve a three attribute query that deal with hierarchical as well as sibling relationships

input: Three inverted lists of length n

output: Three inverted lists of length at most n

computation complexity: $O(n)$

B.13 Recursive tree merge algorithm (page 50)

function: By combining the power of *Recursive multiway merge* algorithm and *Tree like merge*, this algorithm will recursively perform three attribute tree merge on the input tree structures

input: Two records of type *pattern-tree* where each record associates with one inverted list and a pointer to the pattern-tree that associates with the next attribute in the original query

output: A number of inverted lists of length at most n corresponding to the query

computation complexity: $O(mn)$ without direct containment and $O(mnl)$ with direct containment, where m is the number of attributes in the query

B.14 Overlapped list merge algorithm (page 66)

function: Given two overlapping inverted lists X and Y. This algorithm will produce two lists $X' \subseteq X$ *and* $Y' \subseteq Y$ having the same property such that all elements in Y' are contained in one or more elements of X' and all elements in X' contain one or more elements of Y'

input: Two overlapping inverted lists of length n

output: Two overlapping inverted lists of length at most n

computation complexity: $O(n)$

Appendix C

Parent-based partition of OALD structures

Region Types	Parent region types:counts			
ALD	comp2gp:3,	compgp:62,	driv2gp:1,	drivgp:12,
	hwdgp:34,	idmgrp:4,	lsen:6,	nsen:51,
	vppgrp:1			
ALT	comp2gp:8,	compgp:101,	driv2gp:53,	drivgp:212,
	hwdgp:197,	idmgrp:5,	lsen:179,	nsen:863
BTY	DEF:1			
CFR	comp2gp:2,	compgp:22,	driv2gp:13,	drivgp:481,
	hwdgp:1121,	isen:4,	lsen:1035,	nsen:3471
COM	ALT:2,	EXA:19,	IPA:247,	IRF:195,
	IRT:8,	MOD:4,	VAR:7,	VPR:1,
	YPR:21,	compar:48,	compgp:1,	driv2gp:1,
	drivgp:4,	ent:1,	hwdgp:30,	ipr:21,
	irr:37,	irrgrp:54,	isen:31,	lsen:37,
	nsen:66,	sengp:1		
COMP	compgp:7671			

```
COMP2   comp2gp:337

DEF     comp2gp:264,  compgp:6578,  driv2gp:427,  drivgp:5962,
        hwdgp:11057,  idmgrp:6170,  isen:7,       lsen:12292,
        nsen:27320,   vppgrp:2457

DRIV    drivgp:12460

DRIV2   driv2gp:1675

EXA     UNO:596,      comp2gp:51,   compgp:1813,  driv2gp:435,
        drivgp:6394,  hwdgp:9783,   idmgrp:5163,  isen:19,
        lsect:83,     lsen:16852,   nsect:421,    nsen:37908,
        vppgrp:3200,

GEO     DEF:1,        UNO:5,        VAR:1,        comp2gp:23,
        compgp:657,   driv2gp:14,   drivgp:174,   hwdgp:952,
        idmgrp:187,   ipr:6,        irr:95,       lsen:208,
        nsen:1047,    vargrp:904,   vppgrp:80

GLS     DEF:1,        EXA:1928,     UNO:1,        comp2gp:1,
        compgp:99,    driv2gp:13,   drivgp:394,   hwdgp:734,
        idmgrp:263,   isen:3,       lsect:2,      lsen:1262,
        nsect:3,      nsen:3550,    vppgrp:99

GRA     ALT:9,        EXA:4,        USE:2,        VAR:22,
        comp2gp:39,   compgp:1059,  driv2gp:511,  drivgp:3329,
        hwdgp:3871,   idmgrp:46,    lsen:2503,    nsen:8889

HOM     hwdgp:2995

HWD     hwdgp:22289

IDM     idmgrp:10549

ILL     comp2gp:2,    compgp:50,    driv2gp:11,   drivgp:89,
        ent:4,        hwdgp:104,    idmgrp:25,    lsen:15,
```

```
        nsen:55,      vppgrp:10

INT     DEF:2,      IPA:1,        comp2gp:3,    compgp:125,
        driv2gp:11,   drivgp:120,   ent:32,       hwdgp:912,
        idmgrp:53,    ipr:1,        irr:1,        isen:23,
        lsen:67,      nsen:367,     vargrp:1,     vppgrp:16

IPA     ipr:27696

IRF     irr:3940

IRL     IRF:1,        irr:1986

IRT     ipr:1002

LET     UNO:2,        lsect:41,     lsen:12635

MOD     ALT:398,      DEF:1,        EXA:2,        GEO:1055,
        GRA:1229,     INT:1,        IRF:5,        IRL:12,
        PAT:794,      REG:775,      RTY:766,      SBJ:115,
        USE:15,       VAR:118

NUM     isen:3253,    nsect:224,    nsen:31806,   vsen:819

PAT     comp2gp:13,   compgp:158,   driv2gp:16,   drivgp:1748,
        hwdgp:3366,   isen:10,      lsen:3111,    nsen:10081,
        vppgrp:5

PNC     IPA:3,        IRF:1,        comp2gp:1,    compgp:56,
        driv2gp:62,   drivgp:239,   hwdgp:134,    idmgrp:20,
        ipr:133,      irrgrp:278,   lsen:49,      nsen:178,
        vargrp:8,     vppgrp:17

POS     comp2gp:138,  compgp:4573,  driv2gp:1643, drivgp:12492,
        hwdgp:21853,  idmgrp:5,     isen:1,       nsen:28

POS2    comp2gp:2,    compgp:95,    driv2gp:11,   drivgp:119
```

```
REG      EXA:2,         GLS:2,         IDM:1,         VAR:5,
         comp2gp:23,    compgp:732,    driv2gp:57,    drivgp:1261,
         hwdgp:3632,    idmgrp:2294,   lsen:1056,     nsen:4748,
         vppgrp:586

RFW      COM:2,         comp2gp:28,    compgp:746,    driv2gp:27,
         drivgp:278,    hwdgp:2162,    idmgrp:3640,   isen:4,
         lsen:655,      nsen:2115,     vppgrp:63

RTY      comp2gp:29,    compgp:855,    driv2gp:28,    drivgp:281,
         hwdgp:2058,    idmgrp:82,     isen:4,        lsen:744,
         nsen:2362,     vppgrp:50

SBJ      comp2gp:7,     compgp:170,    driv2gp:4,     drivgp:106,
         hwdgp:676,     idmgrp:35,     lsen:115,      nsen:898,
         vppgrp:24

SORT     hwdgp:320,     lsen:1

UNO      ent:197

USE      comp2gp:1,     compgp:93,     driv2gp:19,    drivgp:376,
         hwdgp:555,     idmgrp:317,    isen:1,        lsen:471,
         nsen:1233,     vppgrp:128

VAR      vargrp:3158

VPP      vppgrp:3055

VPR      ipr:310

YPR      ipr:2238

bold     COM:101,       DEF:42,        GLS:2,         GRA:1,
         INT:53,        MOD:2,         RFW:1,         USE:42

comp2gp  compgp:161,    drivgp:151,    ent:25
```

```
compar  comp2gp:1,    compgp:9,     driv2gp:2,    drivgp:9,
        hwdgp:15,     isen:8,       lsen:1,       nsen:3

compgp  ent:7671

driv2gp compgp:282,   drivgp:1367,  ent:25

drivgp  ent:12458

ent     none:22283

form    UNO:1003,     lsect:83,     nsect:457

hn      RFW:2574,     UNO:5,        hw:343

hw      RFW:1024

hwdgp   ent:22283

idmgrp  hwdgp:1,      isen:10367

ipr     COM:2,        COMP:2,       DEF:41,       EXA:9,
        GLS:7,        IDM:2,        IRF:3,        MOD:2,
        UNO:67,       USE:2,        VAR:7,        comp2gp:15,
        compgp:332,   driv2gp:595,  drivgp:4884,  hwdgp:21152,
        idmgrp:13,    irr:986,      lsen:3,       nsect:2,
        nsen:43,      vargrp:308

irr     irrgrp:4077,  vargrp:1

irrgrp  comp2gp:8,    compgp:230,   driv2gp:4,    drivgp:548,
        hwdgp:1938,   lsen:12,      nsen:65,      vppgrp:1

isen    comp2gp:1,    compgp:114,   driv2gp:3,    drivgp:510,
        hwdgp:2871,   lsen:1,       sengp:13

italic  CFR:6,        COM:608,      DEF:531,      GEO:1,
        GLS:28,       GRA:64,       ILL:361,      INT:96,
```

```
        IRF:1,        MOD:908,      RTY:7,        UNO:104,
        USE:2180,     VAR:4,        VPP:1,        lsect:16,
        nsect:64

lsect   UNO:8,        nsect:33

lsen    comp2gp:12,   compgp:455,   driv2gp:41,   drivgp:723,
        hwdgp:1408,   idmgrp:919,   nsen:7855,    vppgrp:1222

nsect   UNO:223

nsen    comp2gp:23,   compgp:1097,  driv2gp:114,  drivgp:5166,
        hwdgp:25169,  sengp:229,    vppgrp:8

pos     RFW:1

ps      RFW:171

roman   CFR:4,        EXA:26,       IRF:1,        MOD:45,
        RFW:10,       VAR:2,        VPP:5

sengp   hwdgp:69

senlb   sengp:69

smcaps  BTY:1,        COM:59,       DEF:344,      INT:3,
        UNO:4,        USE:18,       nsect:1

sn      RFW:891,      hw:25

sup     COM:29,       DEF:380,      EXA:14,       GLS:1,
        INT:1,        UNO:7,        USE:15,       nsect:19,

vargrp  comp2gp:27,   compgp:625,   driv2gp:41,   drivgp:350,
        hwdgp:1156,   idmgrp:17,    isen:1,       lsen:152,
        nsen:582,     vppgrp:16

vpar    vsen:533
```

```
vppgrp  vpar:1150,    vsen:1825

vsen    compgp:3,     drivgp:314,   hwdgp:652,    lsen:2,
        sengp:13
```

Bibliography

[AB77] H.D. Anderson and P.B. Berra. Minimum cost selection of secondary indexes for formatted files. *ACM TODS*, 2(1):68–90, March 1977.

[BBT92] G. Elizabeth Blake, Tim Bray, and Frank Wm. Tompa. Shortening the *oed*: Experience with a grammer-defined database. *ACM Transactions on Information Systems*, 10(3):213–232, July 1992.

[BCD+95] 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: Overview and proposed sql extension. Technical Report CS-95-25, University of Waterloo, June 1995.

[BCK+94] 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 G. Goos and J. Hartmanis, editors, *Proceeding of the First International Conference, ADB-94*, pages 267–280. Springer-Verlag, June 1994.

[BYN96] Ricardo Baeza-Yates and Gonzalo Novarro. Integrating contents and structure in text retrieval. *SIGMOD RECORD*, 25(1):67–79, March 1996.

[C75] Alfonso. F. Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[CCB94] 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.

[CCB95] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[CM94] Mariano P. Consens and Tova Milo. Optimizing queries on files. *SIGMOD Record*, 23(2):301–312, June 1994.

[Cow89] A. P. Cowie, editor. *Oxford Advanced Learner's Dictionary.* Oxford University Press, fourth edition, 1989.

[DL65] D. R. Davis and A. D. Lin. Secondary key retrival using an ibm 7090-1301 system. *Communications of the ACM*, 8(4):243–246, April 1965.

[DSDT96] Tuong Dao, Ron Sacks-Davis, and James A. Thom. Indexing structured test for queries on containment relationships. In *Proceedings of the 7th Australasian Database Conference*, Melbourne, Australia, January 1996.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–75, March 1985.

[FBY92] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms.* Prentice Hall, 1992.

[GBYS92] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. *New Indices for Text: PAT Trees and PAT arrays*, chapter 5, page 66. In Frakes and Baeza-Yates [FBY92], 1992.

[Ger83] Brenda Gerrie. *Online Information Systens: use and operating characteristics, limitations, and design alternatives.* Information Resources Press, Arlington, Virginia, 1983.

[Gol90] C. F. Goldfarb. *The SGML handbook.* Oxford University Press, 1990.

[Gon84] Gaston H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 1984.

[GT87] Gaston H. Gonnet and Frank Wm. Tompa. Mind your grammer: a new approach to modelling text. In Peter M Stocker and William Kent, editors, *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 339–346. Morgan Kaufmann Publishers Inc., September 1987.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problems with Applicaitons to Structured Text Databases.* PhD thesis, University of Helsinki, November 1992.

[KLMN90] Pekka Kilpeläinen, Greger Lindén, Heikki Mannila, and Erja Nikunen. A structured document database system. In R. Furuta, editor, *Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography*, pages 139–151. Cambridge University Press, September 1990.

[KM93] Pekka Kilpeläinen and Heikki Mannila. Retrieval from hierarchical texts by partial patterns. In Edie Rasmussen Robert Korfhage and Peter Willett, editors, *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 214–222, 1993.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, Inc., 1973.

[LC88] G. S. Liu and H. H. Chen. Parallel merging of lists in database management system. *Information Systems*, 13(4):429, 1988.

[Lef69] David Lefkovitz. *File Structures for On-line Systems.* Spartan Books, N. Y., 1969.

[Loe95] A. Loeffen. Text databases: A survey of text models and systems. *SIGMOD RECORD*, 23(1):97–106, March 1995.

[Lum70] V. Y. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM*, 13(11):660–665, Nov. 1970.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, New York, 1968.

[Sal75] Gerard Salton. *Dynamic Information and Library Processing.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Computer Science Series. McGraw-Hill, NY, 1983.

[ST93] Airi Salminen and Frank W. Tompa. Pat expressions: an algebra for text search. *Acta Linguistica Hungarica*, 41:277–306, 1993.

[ST96] Airi Salminen and Frank Wm. Tompa. Grammars++ for modelling information in text. Centre for The NewOED and Text Research, University of Waterloo, Unpublished, 1996.

[Sta90] Craig Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. In *Proceedings of the Thirteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 413–428, 1990.

[Tom89] Frank Wm. Tompa. What is (tagged) text? In *Fifth Annual Conference of the UW Centre for the Oxford English Dictionary*, pages 81–93, September 1989.

[Ull88] Jeffery D. Ullman. *Principles of database and knowledge-base systems*, volume 2 of *Principles of computer science series.* Computer Science Press, 1988.

[Wie87] Gio Wiederhold. *File Organization for Database Design.* McGraw-Hill Inc., 1987. multi-attribute partial match.

[WMB94] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, 1994.