FORMALIZATION AND COMPARISON OF TWO QUERY EVALUATION METHODS IN VERY LARGE FULL-TEXT DATABASES

By

RODNEY LEE BARNETT

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 1991

FORMALIZATION AND COMPARISON OF TWO QUERY EVALUATION METHODS IN VERY LARGE FULL-TEXT DATABASES

Thesis Approved:

M. Samadzadeh-H.
Thesis Advisor

Dean of the Graduate College

ACKNOWLEDGEMENTS

The author thanks Dr. M. Samadzadeh for his advice, patience, and repeated reviews of this thesis during the extended period in which it was completed. Additional thanks go to TMS, Inc. for permission to use the algorithms and methods described herein and to Art Crotzer for his professional and academic guidance. The author thanks committee member Dr. K. M. George for his reviews of this thesis and his academic advice. Final thanks go to committee member Dr. J. P. Chandler for his participation and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

Computers can access very large quantities of data. The compact disc (CD), that typically holds about one hour of high quality digitized music, can hold in excess of 600 megabytes of data when used as a computer storage medium. (CDs are called CD–ROMs [Compact Disc–Read Only Memory] when used as a computer storage medium.) One can collect a library of data on CD–ROMs much the same way that one can collect music on CDs. With such large amounts of data it is necessary to organize the data so that it can be retrieved efficiently. This organization can be achieved in many different ways depending on the type of data and the type of retrieval desired.

One method of data organization is the full-text database. A full-text database is designed for storage and efficient retrieval of textual information. Textual information is the information typically published in books and periodicals like textbooks, journals, manuals, fictional and nonfictional literature, etc. RESEARCH is a full-text retrieval system created by TMS, Inc. [1], [2]. This system uses the printed book as a model for its design. When a database is opened, the first "page"

of the database is visible on the screen. One could read the database page by page, if desired, by merely scrolling through it.

One of the key capabilities provided by RESEARCH, and other full-text retrieval systems, is searching. RESEARCH provides this capability in the form of Boolean-like queries using words which appear in the database as operands. The user enters a search query. RESEARCH parses the query and builds a syntax tree which it then uses to guide the evaluation of the query.

Query evaluation, in the context of full-text databases, is the process of finding the location(s) in the database which meet the criteria of the query. In RESEARCH, this process uses the syntax tree mentioned above and an automatically generated index which lists the location(s) for the words in the database. The syntax tree helps determine which lists to retrieve from the index and how to combine the lists into a single list of locations which meet the criteria of the query.

The current method of query evaluation used in RESEARCH is called cosequential evaluation. Another method being explored at TMS is called incremental query evaluation. This method evaluates the same queries as the cosequential method and produces the same results; however, it takes a more holistic approach to the evaluation process. The incremental evaluation method uses fewer input/output operations, but more CPU operations than the cosequential evaluation method. It also seems more complicated than the cosequential evaluation method due to its

holistic approach—it is difficult to visualize how the method works by analyzing the components in isolation.

One reason to study the incremental query evaluation method is that it may evaluate queries more efficiently than the cosequential query evaluation method. Other methods of optimizing query evaluation have been studied. Most of the research in query optimization emphasizes query evaluation in relational database systems; however, some of the methods can be applied to full-text retrieval systems. [11] gives a survey of general query optimization methods. [8], [10], [13], [14], [16], and [17] describe particular methods of optimizing query evaluation. One common approach is to restate the query to minimize redundancy within the query or to minimize secondary storage access costs. [7], [9], and [18] focus on the optimization of access to secondary storage.

Another form of query optimization being studied is increasing the effectiveness of the query evaluation process in terms of precision (a measure of the number of documents in the answer set which are relevant) and recall (a measure of the number of relevant documents found). [15] defines a method of generalizing the search operators. [12] empirically compares several different methods in terms of performance and effectiveness.

In this thesis, the cosequential query evaluation method and the incremental query evaluation method are formalized and the computational differences between the two methods are discussed. Chapter II introduces the concepts of a full-text

retrieval system that are relevant to searching. Chapter III discusses the cosequential and incremental query evaluation methods. Chapter IV presents a formalism for analyzing the two query evaluation methods. Chapter V contains the comparison of the two query evaluation methods. Chapter VI contains the conclusions of this thesis and some areas of future work.

CHAPTER II

FULL-TEXT RETRIEVAL CONCEPTS

2.1 Document Lists

This section discusses document lists, the components of document lists, and the ordering of document lists. A full-text database is composed of a number of documents. These documents are numbered in ascending order starting with one. The set $D = \{ 1, 2, ..., n \}, n > 0$ represents all valid document numbers for a particular full-text database. (The terms database and full-text database will be used interchangeably henceforth.) Each document contains one or more lexical units (i.e., words). The lexical units are numbered by their offset from the beginning of the documents containing them. For each document, the first lexical unit has an offset of one, the second, two, etc. The set $O_d = \{1, 2, ..., |d|\}, d \in D$, represents all valid lexical unit offsets for document number d and $|d|$ is the number of offsets contained in document number d.

A point on notation is in order here. The symbol d stands for both a natural number (a document number) and a set (a document comprised of lexical units). Hence, it is meaningful to refer to both a document number d in a list of documents

and the size of document number d or $|d|$. Example 1, which follows shortly, clarifies this point.

Combining a document number d and a lexical unit offset o into an ordered pair (d,o) allows each lexical unit to be identified uniquely. Such an ordered pair is called a database location. A special database location (∞,∞) is used as an end-of-list marker. The value of ∞ is chosen so that $\infty > d$ and $\infty > o$ for all document numbers d and lexical unit offsets o. The set $L = \{ (d,o) \mid d \in D \text{ and } o \in O_d \} \cup \{(\infty,\infty)\}$ represents all database locations for a database.

A list of database locations which includes (∞,∞) is called a document list. (This is somewhat of a misnomer since the list also contains lexical unit offsets.) The set $DL = \{ dl \cup (\infty,\infty) \mid dl \subseteq L \}$ represents all document lists for a particular database. Note that $\{ (\infty,\infty) \}$ is the empty document list. It is produced by the query evaluation algorithms when a lexical unit that does not appear in the database is used in a search query or when there are no locations that satisfy the criteria of a search operator in a query.

Many of the uses of document lists presented here assume an ordering of the database locations within the document lists. This ordering is defined as follows. Let $l_1, l_2 \in L$ where $l_1 = (d_1,o_1)$ and $l_2 = (d_2,o_2)$, then

(i) $l_1 < l_2$ if and only if either $d_1 < d_2$, or $d_1 = d_2$ and $o_1 < o_2$;
(ii) $l_1 = l_2$ if and only if $d_1 = d_2$ and $o_1 = o_2$; and, by exclusion,
(iii) $l_1 > l_2$ if and only if either $d_1 > d_2$, or $d_1 = d_2$ and $o_1 > o_2$.

TABLE I

A SAMPLE FULL-TEXT DATABASE

Document Number	Lexical Unit Offset															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	–	–	–	–	w	–	x	–	–	–	–	–	–	–	w	–
2	–	–	w	–	–	–										
3	x	y	z	w	x	–	–	–	–							
4	–	x	–	–	–	–	z	–	–	–	–	–	–	–		
5	w	–	–	–	–	–	–	–	y	–	w	–	–	–	–	–
6	x	–	–	–	y	–	–	–	–	–	–	–				
7	–	w	z	–												
8	–	–	–	–	–	–	–	y	–	–	–					
9	–	–	–	x	–	–	–	–	–	–	–	–	–	–	–	
10	–	–	–	–	–	–	–	–	–	–	–	–	–			

Example 1

TABLE I contains a sample contrived full-text database. Letters w, x, y, and z each represent the position of a particular lexical unit and "–" represents the position of other lexical units. A blank position indicates that no lexical unit exists at that location. For this sample database, D = { 1, 2, ..., 10 }, there are ten sets of lexical unit offsets O_1 = { 1, 2, ..., 16 }, ..., O_{10} = { 1, 2, ..., 13 }, and L= { (1,1), ..., (1,16), (2,1), ..., (2,6), ..., (10,1), ..., (10,13), (∞,∞) }. Four sample document lists, each of which contains all of the locations of a particular lexical unit, are

dl_w = { (1,5), (1,15), (2,3), (3,4), (5,1), (5,11), (7,2), (∞,∞) },
dl_x = { (1,7), (3,1), (3,5), (4,2), (6,1), (9,4), (∞,∞) },

dl_Y = { (3,2), (5,9), (6,5), (8,8), (∞,∞) }, and
dl_Z = { (3,3), (4,7), (7,3), (∞,∞) }.†

2.2 Boolean Search Queries

Boolean search queries contain lexical units, which actually appear in the database, intermingled with search operators to form an infix search expression. These queries allow the user to indicate which documents he/she desires by describing the lexical units that these documents should contain and the interrelationships among those lexical units.

Full-text retrieval systems use several different Boolean query operators. Most of the operators are based on Boolean logic which is the namesake of these queries. For example, the search query "income and tax" would result in a list of documents in the database which contain both the lexical unit "income" and the lexical unit "tax."

Only the AND, OR, and AND NOT search operators are discussed in this thesis. The formal definitions of these operators follow. These definitions assume that $dl_1, dl_2 \in DL$.

---

† Note that the document lists are listed in increasing order according to the ordering relation. This will be the convention henceforth.

AND : DL × DL → DL is defined by AND (dl_1, dl_2) = { (d_i,o_i) | (d_i,o_i) ∈ dl_1 and ∃ (d_j,o_j) ∈ dl_2 such that $d_i = d_j$ } ∪ { (d_j,o_j) | (d_j,o_j) ∈ dl_2 and ∃ (d_i,o_i) ∈ dl_1 such that $d_j = d_i$ }

OR : DL × DL → DL is defined by OR (dl_1, dl_2) = $dl_1 \cup dl_2$

AND NOT : DL × DL → DL is defined by AND NOT (dl_1, dl_2) = { (d_i,o_i) | (d_i,o_i) ∈ dl_1 and ∄(d_j,o_j) ∈ dl_2 such that $d_i = d_j$} ∪ { (∞,∞) }

Example 2

In the following examples, dl_w, dl_x, dl_y, and dl_z are as defined in Example 1.

AND NOT (dl_w, dl_x) = AND NOT ({ (1,5), (1,15), (2,3), (3,4), (5,1), (5,11), (7,2), (∞,∞)}, { (1,7), (3,1), (3,5), (4,2), (6,1), (9,4), (∞,∞) }) = { (2,3), (5,1), (5,11), (7,2), (∞,∞) }

OR (dl_y, dl_z) = OR ({ (3,2), (5,9), (6,5), (8,8), (∞,∞) }, { (3,3), (4,7), (7,3), (∞,∞) }) = {(3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8), (∞,∞) }

AND (AND NOT (dl_w, dl_x), OR (dl_y, dl_z)) = {(5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞)}

2.3 Syntax Tree

A syntax tree results from parsing a search query. The syntax tree is used to guide the query evaluation process in both cosequential and incremental evaluation methods. This section provides a formal definition of the syntax tree.

Aho and Ullman [3] provide the following definition for a tree.†

> A tree T is a directed graph G = (A, R), where A is the set of nodes and R is the set of edges, with a specified node *r* in A called the root such that
> - *r* has in-degree 0,
> - All other nodes of T have in-degree 1, and
> - Every node in A is accessible from *r*.

As an extension to the definition of a tree, we can define a binary tree BT as a tree T such that the out-degree of any node is at most two. Each direct descendent of a node in a BT is designated either as the left child or the right child of that node such that each node has at most one left child and one right child. Unless otherwise specified, we will refer to a binary tree as a tree.

A syntax tree ST = (BT, *r*, *f*) is a labeled binary tree BT with root *r* such that a labeling function *f* maps each node of the BT to a type $t \in P$, where P = {AND, OR, AND NOT, LU} is a set of node types. The types AND, OR, and AND NOT associated with a node indicate that the node is an internal node labeled with the appropriate search operator. The type LU indicates that the node is a leaf node associated with the document list for a particular lexical unit that was specified in the search query (see the definition of DLIST below).‡

---

† Terminology relating to trees and directed graphs corresponds to that used in [3].

‡ Note that a node *a* for which *f* (*a*) = AND will be referred to as an AND node. A node whose type is OR or AND NOT will be treated analagously. A node whose type is LU will be referred to as a leaf node.

Several functions are used in the ensuing discussion as a notational convenience. The definitions given below assume that BT = (A, R) is a binary tree with root node r and $a, b \in A$.

LEFT : A → A is defined as LEFT $(a) = b$ where b is the left child of a if a has a left child and *nil* otherwise.

RIGHT : A → A is defined as RIGHT $(a) = b$ where b is the right child of a if a has a right child and *nil* otherwise.

PARENT : A → A is defined as PARENT $(a) = b$ where b is the direct ancestor (immediate predecessor) of a if $a \neq r$ and *nil* if $a = r$.

SIB : A → A is defined as

$$\text{SIB}(a) = \begin{cases} nil, & \text{if } a = r \\ \text{LEFT}(\text{PARENT}(a)), & \text{if RIGHT}(\text{PARENT}(a)) = a \\ \text{RIGHT}(\text{PARENT}(a)), & \text{if LEFT}(\text{PARENT}(a)) = a \end{cases}$$

DLIST : A → DL is defined as DLIST (a) = dl, such that dl = $\emptyset$ if $f(a) \neq$ LU and dl is the document list associated with node a otherwise.†

---

† An implementation of either the incremental or cosequential search evaluation method would probably make this association indirect by associating a string containing a lexical unit with each node of type LU. The lexical unit would then be used as a key to retrieve a document list from an index containing one such document list for each lexical unit in the database. The document list would contain the locations of all occurrences of the lexical unit in the database.

Example 3

Consider the search query "(w and not x) and (y or z)." The syntax tree ST for this query would be ST = (BT, r, f), BT = $(\{r, u, v, w, x, y, z\}, \{ (r,u), (r,v), (u,w), (u,x), (v,y), (v,z) \})$ with $f(r)$ = AND, $f(u)$ = AND NOT, $f(v)$ = OR, and $f(w)$ = $f(x)$ = $f(y)$ = $f(z)$ = LU. Also, u, w, and y are designated as left children and v, x, and z are designated as right children.

TABLE II gives the values of the LEFT, RIGHT, PARENT, SIB, and DLIST functions, which were defined earlier in this section, for the syntax tree ST. Note that the document lists dl_w, dl_x, dl_y, and dl_z are as defined in Example 1.

TABLE II

VALUES OF THE SYNTAX TREE FUNCTIONS FOR THE QUERY IN EXAMPLE 3

Node	Function				
	LEFT	RIGHT	PARENT	SIB	DLIST
r	u	v	*nil*	*nil*	*nil*
u	w	x	r	v	*nil*
v	y	z	r	u	*nil*
w	*nil*	*nil*	u	x	dl_w
x	*nil*	*nil*	u	w	dl_x
y	*nil*	*nil*	v	z	dl_y
z	*nil*	*nil*	v	y	dl_z

This chapter has defined the concepts which are shared by the cosequential query evaluation method and the incremental query evaluation method. These concepts are necessary for further discussion of these methods. The concepts defined include document lists which represent sets of lexical units within a full-text database, Boolean search queries which identify a set of lexical units to retrieve from a full-text database, and syntax trees which represent Boolean search queries. The next chapter uses these concepts to discuss the algorithms used in the two query evaluation methods.

CHAPTER III

QUERY EVALUATION ALGORITHMS

Both the cosequential and the incremental query evaluation methods traverse the syntax tree during the evaluation process. The cosequential method performs a single postorder traversal while the incremental method performs a large number of partial traversals. Each method uses a different algorithm for each of the four types of nodes in the syntax tree and when a node is encountered, the algorithm for that type of node is invoked to perform the next step in the evaluation process. Because a syntax tree can contain more than one node of any one type, there does not exist a one-to-one relationship between the nodes and algorithms; hence, the relationship between nodes and algorithms will be referred to as an association.

This chapter describes both of the query evaluation methods and the algorithms of which they are comprised. The algorithms presented here are based on algorithms developed by the technical staff at TMS, Inc. [1], [2].

3.1 Cosequential Query Evaluation

In the cosequential query evaluation method, each of the search operators takes two input document lists and produces a single output document list. The input

document lists are produced by another search operator or are retrieved from some type of inverted index file which allows access to the document lists via a lexical unit key. The two input document lists are processed sequentially and in combination to produce the output document list. This type of combined sequential operation is called a cosequential operation [6]. Using this definition of the term cosequential, the algorithms which implement the search operators are called cosequential query evaluation algorithms and the search evaluation process is called cosequential query evaluation. This section discusses the cosequential query evaluation algorithms.

Each cosequential query evaluation algorithm requires two node identifiers as input. These node identifiers identify the children of the node which is associated with the current invocation of the cosequential algorithm and allow the algorithm to invoke (recursively) the algorithms associated with the children of the node to produce the two complete document lists which the algorithm then processes to produce a third complete document list. In the algorithms presented in this section, the invocation of an algorithm is indicated by the use of the function GetDList : $A^3 \rightarrow DL$. The value of GetDList (a, b, c), $a, b, c \in A$ is the document list produced by whatever algorithm is associated with node a given the children b and c of a as input.

ALGORITHM 1.

Cosequential evaluation of the OR search operator.

Input. The input consists of two node identifiers $a, b \in A$.

Output. The output is a single document list dl containing all locations in the document lists produced by the algorithms associated with the input nodes which meet the criteria of the OR search operator.

Method.

Step 1. Let $dl_a = \{l_1, l_2, ..., l_r\}$ = GetDList (a, LEFT (a), RIGHT (a)) with $l_r = (\infty,\infty)$.

Step 2. Let $dl_b = \{m_1, m_2, ..., m_s\}$ = GetDList (b, LEFT (b), RIGHT (b)) with $m_s = (\infty,\infty)$.

Step 3. Initialize i to 1, j to 1, and dl to $\emptyset$.

Step 4. If $l_i < m_j$, let dl = dl $\cup \{l_i\}$ and $i = i + 1$, repeat *Step 4* until $l_i \geq m_j$.

Step 5. If $l_i > m_j$, let dl = dl $\cup \{m_j\}$ and $j = j + 1$, repeat *Step 5* until $l_i \leq m_j$.

Step 6. If $l_i \neq m_j$, go to *Step 4*.

Step 7. Let dl = dl $\cup \{l_i\}$.

Step 8. If $l_i \neq (\infty,\infty)$, let $i = i + 1$ and $j = j + 1$. Go to *Step 4*.

Step 9. End of algorithm.

Example 4

TABLE III illustrates an application of Algorithm 1 by showing the sequence of steps which affect l_i, m_j, and dl during the cosequential evaluation of the node v in the syntax tree given in Example 3 in Chapter II. To reduce clutter, steps that do not affect l_i, m_j, or dl are not shown.

ALGORITHM 2.

Cosequential evaluation of the AND NOT search operator.

Input. The input consists of two node identifiers $a, b \in$ A.

Output. The output is a single document list dl containing all locations in the document lists produced by the algorithms associated with the input nodes which meet the criteria of the AND NOT search operator.

TABLE III

SAMPLE INVOCATION OF ALGORITHM 1

Step	l_i	m_j	dl
3	(3,2)	(3,3)	Ø
4	(5,9)		{ (3,2) }
5		(4,7)	{ (3,2), (3,3) }
5		(7,3)	{ (3,2), (3,3), (4,7) }
4	(6,5)		{ (3,2), (3,3), (4,7), (5,9) }
4	(8,8)		{ (3,2), (3,3), (4,7), (5,9), (6,5) }
5		(∞,∞)	{ (3,2), (3,3), (4,7), (5,9), (6,5), (7,3) }
4	(∞,∞)		{ (3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8) }
7			{ (3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8), (∞,∞) }

Method.

Step 1. Let $dl_a = \{l_1, l_2, ..., l_r\}$ = GetDList (a, LEFT (a), RIGHT (a)) with $l_i = (d_i,o_i)$ for $1 \le i \le r$ and $l_r = (\infty,\infty)$.

Step 2. Let $dl_b = \{m_1, m_2, ..., m_s\}$ = GetDList (b, LEFT (b), RIGHT (b)) with $m_j = (e_j,p_j)$ for $1 \le j \le s$ and $m_s = (\infty,\infty)$.

Step 3. Initialize i to 1, j to 1, and dl to Ø.

Step 4. If $d_i < e_j$, let dl = dl $\cup \{l_i\}$ and $i = i + 1$, repeat *Step 4* until $d_i \ge e_j$.

Step 5. If $d_i = \infty$, go to *Step 9*.

Step 6. If $d_i > e_j$, let $j = j + 1$, repeat *Step 6* until $d_i \le e_j$.

Step 7. If $d_i = e_j$, let $i = i + 1$, repeat *Step 7* until $d_i > e_j$.

Step 8. Go to *Step 4*.

Step 9. Let dl = dl $\cup \{(\infty,\infty)\}$.

Step 10. End of algorithm.

Example 5

TABLE IV illustrates an application of Algorithm 2 by showing the sequence of steps which affect l_i, m_j, and dl during the cosequential evaluation of the node u in the syntax tree given in Example 3 of Chapter II. To reduce clutter, steps that do not affect l_i, m_j, or dl are not shown.

TABLE IV

SAMPLE INVOCATION OF ALGORITHM 2

Step	l_i	m_j	dl
3	(1,5)	(1,7)	Ø
7	(1,15)		
7	(2,3)		
6		(3,1)	
4	(3,4)		{ (2,3) }
7	(5,1)		
6		(3,5)	
6		(4,2)	
6		(6,1)	
4	(5,11)		{ (2,3), (5,1) }
4	(7,2)		{ (2,3), (5,1), (5,11) }
6		(9,4)	
4	(∞,∞)		{ (2,3), (5,1), (5,11), (7,2) }
9			{ (2,3), (5,1), (5,11), (7,2), (∞,∞) }

ALGORITHM 3.

Cosequential evaluation of the AND search operator.

Input. The input consists of two node identifiers $a, b \in A$.

Output. The output is a single document list dl containing all locations in the document lists produced by the algorithms associated with the input nodes which meet the criteria of the AND search operator.

Method.

Step 1. Let $dl_a = \{l_1, l_2, ..., l_r\}$ = GetDList (a, LEFT (a), RIGHT (a)) with $l_i = (d_i,o_i)$ for $1 \le i \le r$ and $l_r = (\infty,\infty)$.

Step 2. Let $dl_b = \{m_1, m_2, ..., m_s\}$ = GetDList (b, LEFT (b), RIGHT (b)) with $m_j = (e_j,p_j)$ for $1 \le j \le s$ and $m_s = (\infty,\infty)$.

Step 3. Initialize i to 1, j to 1, and dl to $\emptyset$.

Step 4. If $d_i < e_j$, let $i = i + 1$, repeat *Step 4* until $d_i \ge e_j$. Go to *Step 10.*

Step 5. If $d_i > e_j$, let $j = j + 1$, repeat *Step 5* until $d_i \le e_j$. Go to *Step 10.*

Step 6. Let $d = d_i$.

Step 7. If $l_i < m_j$, let dl = dl $\cup \{l_i\}$ and $i = i + 1$. Otherwise, if $l_i > m_j$, let dl = dll $\cup \{m_j\}$ and $j = j + 1$. Otherwise, let dl = dl $\cup \{l_i\}$, $i = i + 1$, and $j = j + 1$.

Step 8. If $d_i = d$ and $e_j = d$, go to *Step 7.*

Step 9. If $d_i = d$, let $dl_t = \{ l \mid l \in dl_a$ and $l \ge l_i$ and $d = d_i \}$ and dl = dl $\cup$ dl_t and $i = i + |dl_t|$. Otherwise, if $e_j = d$, let $dl_t = \{l \mid l \in dl_b$ and $l \ge m_j$ and $d = e_j\}$, dl = dl $\cup$ dl_t, and $j = j + |dl_t|$.

Step 10. If $l_i \ne (\infty,\infty)$ and $m_j \ne (\infty,\infty)$, go to *Step 4.*

Step 11. dl = dl $\cup \{(\infty,\infty)\}$.

Step 12. End of algorithm.

Example 6

TABLE V illustrates an application of Algorithm 3 by showing the sequence of steps which affect l_i, m_j, and dl during the cosequential evaluation of the node r in the syntax tree given in Example 3 of Chapter II. To reduce clutter, steps that do not affect l_i, m_j, or dl are not shown.

TABLE V

SAMPLE INVOCATION OF ALGORITHM 3

Step	l_i	m_j	dl
3	(2,3)	(3,2)	Ø
4	(5,1)		
5		(3,3)	
5		(4,7)	
5		(5,9)	
7	(5,11)		{ (5,1) }
7		(6,5)	{ (5,1), (5,9) }
9	(7,2)		{ (5,1), (5,9), (5,11) }
5		(7,3)	
7	(∞,∞)		{ (5,1), (5,9), (5,11), (7,2) }
9		(8,8)	{ (5,1), (5,9), (5,11), (7,2), (7,3) }
11			{ (5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞) }

3.2 Incremental Query Evaluation

Incremental evaluation is an alternative method of evaluating a search query. It produces locations in the final document list one at a time (i.e., incrementally),

rather than all at once by the evaluation of the operator at the root of the syntax tree, as in the case of cosequential evaluation.

The syntax tree used by the cosequential evaluation is used to guide the incremental evaluation also; however, the order which the tree is traversed is not strictly postorder, nor do the incremental evaluation algorithms produce complete document lists when they are called. A postorder traversal is completed for each database location which meets the criteria of the query; however, there is typically much backtracking before the traversal is completed. When an operator has at least one database location from each of its operands, it starts processing until it has a single database location which meets its criteria. This processing may require additional database locations from one or both of its operands. When an operator has a database location which meets its criteria, it passes that location to the operator which is its parent in the syntax tree. When the operator which is at the root of the syntax tree has a database location that meets its criteria, the number (i.e., the database location) is added to the final document list for the query.

Incremental evaluation would be nothing more than a complicated way to do the cosequential evaluation were it not for one additional feature. When an operator calls one of its operands for additional database locations, it passes a database location to it called the minimum return value. The operand must produce a database location which is at least as large as this minimum return value. This allows the

operand to skip all the database locations that are not large enough to pass the requirement on to its operands.

The incremental search evaluation algorithms (except Algorithm 4, the driver algorithm) return at most one database location. This location meets the criteria of the search operator that the particular algorithm evaluates. There are typically many database locations which meet the criteria of a search operator. The location returned by a particular invocation of an incremental algorithm is the smallest one which is at least as large as an input parameter called the minimum return value (i.e., the value returned must be greater than or equal to the minimum return value).

Algorithm 4 drives the incremental evaluation process by repeatedly invoking the algorithm associated with the root node of the syntax tree until (∞,∞) is returned. The location or value (1,1) is used as the minimum return value in the first invocation and the value $(d,o+1)$ is used in all subsequent invocations where (d,o) is the location returned by the previous invocation.

Note that $(d,o+1) \notin \mathrm{L}$ if $o = |d|$ (i.e., o represents the last lexical unit in document number d). This is one of the reasons that minimum return values are elements of a superset of L denoted L′. Another reason is that Algorithm 6, which evaluates the AND search operator, can use the value $(\infty,1)$ as a minimum return value when invoking a child. So, a set of potential minimum return values would be $\mathrm{L}' = \mathrm{L} \cup \{ (d,o+1) \mid (d,o) \in \mathrm{L} \text{ and } o = |d| \} \cup \{ (\infty,1) \}$.

The invocation of a child in the incremental algorithms is indicated by the use of a function GetNext : A × L′ → L, where A is the set of nodes in the syntax tree, L′ is the set of minimum return values, and L is the set of database locations. The value of GetNext (a, l_m), with $a \in$ A and $l_m \in$ L′, is the return value l_r from the invocation of the algorithm associated with node a with l_m and all required outputs from the previous invocation used as inputs. For example, given the syntax tree of Example 3, GetNext (r, (5,10)) would cause Algorithm 7 (the AND algorithm) to be invoked with the minimum return value of (5,10). In addition, the node identifiers for the children of r (i.e., u and v) and the database locations output from the previous invocation of Algorithm 7 (i.e., l_a, l_b, and l_r) would be provided as input to the current invocation.

ALGORITHM 4.

Incremental driver algorithm.

Input. The input is a syntax tree, ST = (BT, r, f), BT = (A, R) representing the query to be evaluated.

Output. The output is a document list dl that is the result of the evaluation of the query represented by the syntax tree ST.

Method.

Step 1. Initialize dl to Ø and l_m to (1,1).

Step 2. Let l = GetNext (r, l_m).

Step 3. Let dl = dl ∪ { l }.

Step 4. If $l = (\infty,\infty)$, go to *Step 7.*

Step 5. Let $l_m = (d,o+1)$.† /* Update the minimum return value. */

Step 6. Go to *Step 2.*

Step 7. End of algorithm.

The incremental algorithms for AND, OR, and AND NOT are presented below.

ALGORITHM 5.

Incremental evaluation of the OR search operator.

Input. The input consists of three database locations l_a, l_b, $l_m \in$ L, where l_a and l_b are outputs of the previous invocation of this algorithm or (-1,-1), if this is the first invocation of the algorithm, and l_m is the minimum return value, and two node identifiers $a, b \in$ A that identify the left and right children of the node in the syntax tree which is associated with the current invocation of this algorithm.

Output. The output consists of three database locations l_a, l_b, $l_r \in$ L, where l_a and l_b are values that should be input to the next invocation of this algorithm and l_r is the return value of the current invocation of this algorithm.

Method.

Step 1. If $l_a < l_m, l_a =$ GetNext $(a, l\,m)$. If $l_b < l_m, l_b =$ GetNext (b, l_m).

Step 2. If $l_a < l_b, l_r = l_a$; otherwise, $l_r = l_b$.

Step 3. End of algorithm.

---

† Note that each instance of l is an ordered pair (d,o), so *Step 5* is incrementing the lexical unit offset portion of l and assigning that value to the next minimum return value.

Example 7

TABLE VI illustrates the incremental evaluation of the OR search operator in the query given in Example 3 in Chapter II by showing the input and output values of all of the invocations of Algorithm 5 in that evaluation process. Note that the values of the input node identifiers a and b are not shown in the table; in this case, they are y and z, respectively.

TABLE VI

SAMPLE EXECUTION TRACE OF ALGORITHM 5

	Input			Output		
Invocation	l_a	l_b	l_m	l_a	l_b	l_r
1	(-1,-1)	(-1,-1)	(1,1)	(3,2)	(3,3)	(3,2)
2	(3,2)	(3,3)	(5,1)	(5,9)	(7,3)	(5,9)
3	(5,9)	(7,3)	(5,10)	(6,5)	(7,3)	(6,5)
4	(6,5)	(7,3)	(7,1)	(8,8)	(7,3)	(7,3)
5	(8,8)	(7,3)	(7,4)	(8,8)	(∞,∞)	(8,8)
6	(8,8)	(∞,∞)	(∞,1)	(∞,∞)	(∞,∞)	(∞,∞)

ALGORITHM 6.

Incremental evaluation of the AND NOT search operator.

Input. The input consists of three database locations l_a, l_b, $l_m \in$ L, where l_a and l_b are outputs of the previous invocation of this algorithm or (-1,-1), if this is the first invocation of the algorithm, and l_m is the minimum return value, and two node identifiers a, $b \in$ A that identify the left and right children of the node in the syntax tree which is associated with the current invocation of this algorithm.

Output. The output consists of three database locations l_a, l_b, $l_r \in$ L, where l_a and l_b are values which should be input to the next invocation of this algorithm and l_r is the return value of the current invocation of this algorithm.

Method.

Step 1. If $l_a < l_m$, l_a = GetNext (a, l_m). If $l_b < l_m$, l_b = GetNext (b, l_m).

Step 2. If $l_a = (\infty,\infty)$, $l_r = l_a$. Go to *Step 7.*

Step 3. If $d_a < d_b$, $l_r = l_a$. Go to *Step 7.*

Step 4. If $d_a > d_b$, l_b = GetNext $(b, (d_a,1))$

Step 5. If $d_a = d_b$, l_a = GetNext $(a, (d_a+1,1))$.

Step 6. Go to *Step 2.*

Step 7. End of algorithm.

Example 8

TABLE VII illustrates the incremental evaluation of the AND NOT search operator in the query given in Example 3 in Chapter II by showing the input and out-

TABLE VII

SAMPLE EXECUTION TRACE OF ALGORITHM 6

	Input			Output		
Invocation	l_a	l_b	l_m	l_a	l_b	l_r
1	(-1,-1)	(-1,-1)	(1,1)	(2,3)	(3,1)	(2,3)
2	(2,3)	(3,1)	(3,1)	(5,1)	(6,1)	(5,1)
3	(5,1)	(6,1)	(5,2)	(5,11)	(6,1)	(5,11)
4	(5,11)	(6,1)	(5,12)	(7,2)	(9,4)	(7,2)
5	(7,2)	(9,4)	(7,3)	(∞,∞)	(9,4)	(∞,∞)

put values of all of the invocations of Algorithm 6 in that evaluation process. Note that the values of the input node identifiers a and b are not shown in the table; in this case, they are w and x, respectively.

ALGORITHM 7.

Incremental evaluation of the AND search operator.

Input. The input consists of four database locations $l_a, l_b, l_r, l_m \in$ L, where l_a and l_b are outputs of the previous invocation of this algorithm or (-1,-1), if this is the first invocation of the algorithm, l_r is the return value of the previous invocation or (-1,-1), if this is the first invocation algorithm, and l_m is the minimum return value, and two node identifiers $a, b \in$ A that identify the left and right children of the node in the syntax tree which is associated with the current invocation of this algorithm.

Output. The output consists of three database locations $l_a, l_b, l_r \in$ L, where all three values should be input to the next invocation of this algorithm and l_r is the return value of the current invocation of this algorithm.

Method.†

Step 1. If $l_a < l_m$, l_a = GetNext (a, l_m). If $l_b < l_m$, l_b = GetNext (b, l_m).

Step 2. If $d_a = d_r$ and $l_a \leq l_b$, $l_r = l_a$. Go to *Step 8*.

Step 3. If $d_b = d_r$ and $l_a > l_b$, $l_r = l_b$. Go to *Step 8*.

Step 4. If $d_a = d_b$, $l_r = \min \{ l_a, l_b \}$. Go to *Step 8*.

Step 5. If $d_a < d_b$, l_a = GetNext $(a, (d_b,1))$.

Step 6. If $d_a > d_b$, l_b = GetNext $(b, (d_a,1))$.

Step 7. Go to *Step 4*.

Step 8. End of algorithm.

---

† Note that instances of l_a and l_b are ordered pairs (d_a,o_a) and (d_b,o_b) respectively, so statements that modify l_a and/or l_b also modify d_a and/or d_b.

Example 9

TABLE VIII illustrates the incremental evaluation of the AND search operator in the query given in Example 3 in Chapter II by showing the input and output values of all of the invocations of Algorithm 7 in that evaluation process. Note that the values of the input node identifiers *a* and *b* are not shown in the table; in this case, they are *u* and *v*, respectively.

TABLE VIII

SAMPLE EXECUTION TRACE OF ALGORITHM 7

	Input				Output		
Invocation	l_a	l_b	l_m	l_r	l_a	l_b	l_r
1	(-1,-1)	(-1,-1)	(1,1)	(-1,-1)	(5,1)	(5,9)	(5,1)
2	(5,1)	(5,9)	(5,2)	(5,1)	(5,11)	(5,9)	(5,9)
3	(5,11)	(5,9)	(5,10)	(5,9)	(5,11)	(6,5)	(5,11)
4	(5,11)	(6,5)	(5,12)	(5,11)	(7,2)	(7,3)	(7,2)
5	(7,2)	(7,3)	(7,3)	(7,2)	(∞,∞)	(7,3)	(7,3)
6	(∞,∞)	(7,3)	(7,4)	(7,3)	(∞,∞)	(∞,∞)	(∞,∞)

This chapter defined the cosequential query evaluation method and the incremental query evaluation method by presenting the algorithms which implement each of these methods. For each method, one algorithm was presented for each of the three search operators (the AND operator, the AND NOT operator, and the OR operator). An additional driver algorithm was presented for the incremental evalua-

tion method. The next chapter formalizes the two evaluation methods defined by the algorithms in this chapter.

CHAPTER IV

FORMALIZATION OF THE EVALUATION METHODS

The interaction among the search operators in the cosequential and incremental evaluation methods differ. In the cosequential evaluation method, evaluation of an operator has no effect on the output of the search operator(s) which are its descendents in the syntax tree corresponding to the search query. In contrast, evaluation of an operator can effect the outputs of its descendents in the incremental evaluation method. This means that a comparison of the two evaluation methods must examine the search operators in the context of the entire query rather than in isolation. This chapter contains such an analysis.

4.1 Cosequential Evaluation Algorithms

In the cosequential evaluation method, the search operators produce a complete document list given two complete document lists as input. The isolated nature of the operators in this method, as opposed to the incremental evaluation method, makes it quite easy to describe the result of processing a node in the syntax tree during query evaluation. The definition of OUT below is such a description. The value of OUT, when given the root node of a syntax tree, is the document list result-

ing from the evaluation of the query which was parsed to obtain that syntax tree. When given any other node in a syntax tree, the value of OUT will be the partial document list resulting from the evaluation of that node and all descendents of that node.

Let ST = (BT, r, f), BT = (A, R) be a syntax tree and $a \in$ A, then

OUT : A → DL is defined as

$$\text{OUT}(a) = \begin{cases} \text{DLIST}(a), & \text{if } f(a) = \text{LU} \\ \text{AND}(\text{OUT}(\text{LEFT}(a)), \text{OUT}(\text{RIGHT}(a))), & \text{if } f(a) = \text{AND} \\ \text{OR}(\text{OUT}(\text{LEFT}(a)), \text{OUT}(\text{RIGHT}(a))), & \text{if } f(a) = \text{OR} \\ \text{AND NOT}(\text{OUT}(\text{LEFT}(a)), \text{OUT}(\text{RIGHT}(a))), & \text{if } f(a) = \text{AND NOT} \end{cases}$$

Example 10

This example illustrates OUT by giving the values for the nodes in the sample syntax tree given in Example 3 in Chapter II. The contents of the resulting sets can be seen in Examples 1 and 2 in Chapter II.

$\text{OUT}(z) = \text{DLIST}(z) = \text{dl}_z$
$\text{OUT}(y) = \text{DLIST}(y) = \text{dl}_y$
$\text{OUT}(x) = \text{DLIST}(x) = \text{dl}_x$
$\text{OUT}(w) = \text{DLIST}(w) = \text{dl}_w$
$\text{OUT}(v) = \text{OR}(\text{OUT}(y), \text{OUT}(z)) = \text{OR}(\text{dl}_y, \text{dl}_z)$
$\text{OUT}(u) = \text{AND NOT}(\text{OUT}(w), \text{OUT}(x)) = \text{AND NOT}(\text{dl}_w, \text{dl}_x)$
$\text{OUT}(r) = \text{AND}(\text{OUT}(u), \text{OUT}(v))$
$= \text{AND}(\text{AND NOT}(\text{dl}_w, \text{dl}_x), \text{OR}(\text{dl}_y, \text{dl}_z))$

4.2 Incremental Evaluation Algorithms

In the incremental evaluation method, the search operators interact much more frequently than in the cosequential method. The increased interaction allows the incremental evaluation method to use a minimum return value as described in Section 2 of Chapter III. This minimum return value allows the constraints from the search operator to be "passed around" the syntax tree so that those database locations that do not meet the constraints can be eliminated more quickly than it is possible in the cosequential evaluation method. On the other hand, this increased interaction makes the output of each operator more difficult to define and analyze.

4.2.1 Incremental Support Algorithms

Before examining the incremental evaluation method, a brief digression is necessary to define three functions, G, GE, and LV, which will be useful in that examination. Informally, G and GE accept two document lists, each element of which meets two criteria, and extract a set of locations from the first list. The first criterion is that each element must be greater than, or greater than or equal to in the case of GE, some location *l* in the second document list. The second criterion is that each element must be the smallest element which meets the first criterion for *l*. The function LV (short for loop values) returns a set containing all locations, which are between pairs of locations in the second and third input sets, from the first input set. These locations are within the range of the loops of the incremental AND and AND

NOT algorithms. These functions play a central role in the description of the results of the incremental search operators. The formal definitions of G, GE, and LV follow.

G : DL × DL → DL is defined as G (dl_1,dl_2) = $\{ l_1 \mid l_1 \in dl_1$ and $l_1 = \min \{ l'_1 \mid l'_1 \in dl_1$ and $l'_1 > l_2$ for some $l_2 \in dl_2 \} \}$.

GE : DL × DL → DL is defined as GE (dl_1,dl_2) = $\{ l_1 \mid l_1 \in dl_1$ and $l_1 = \min \{ l'_1 \mid l'_1 \in dl_1$ and $l'_1 \geq l_2$ for some $l_2 \in dl_2 \} \}$.

LV : DL × DL × DL → DL is defined as LV (dl_1,dl_2,dl_3) = $\{ l_1 \mid l_1 \in dl_1$ and $l_2 < l_1 \leq l_3$, where $l_2 \in dl_2$ and $l_3 \in$ GE (dl_3,$\{ l_2 \}$) $\}$.

Example 11

This example illustrates G, GE, and LV by giving the values of these functions for some of the invocations in Example 15.

G ({ (3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8), (∞,∞) }, { (2,3), (5,1), (5,11), (7,2), (∞,∞) }) = { (3,2), (5,9), (6,5), (7,3) }

GE ({ (2,3), (5,1), (5,11), (7,2), (∞,∞) }, { (1,1), (5,2), (5,10), (5,12), (7,3), (7,4) }) = { (2,3), (5,11), (7,2), (∞,∞) }

LV ({3,2), (5,9), (6,5), (7,3) }, { (2,3), (5,11), (7,2), (∞,∞) }, { (5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞) }) = { (3,2) }

4.2.2 AOUT

The output of a node in the incremental evaluation method can be viewed as an adjusted version of the output of a node in the cosequential evaluation method. For this reason, the output of a node in the incremental evaluation method is denoted AOUT. The adjustment is relatively simple: all outputs, which are not at least as large as the minimum return value input to the algorithm, are skipped. This can be stated formally as follows.

Let ST = (BT, r, f), BT = (A, R) be a syntax tree and $a \in$ A, then AOUT : A → DL is defined as AOUT (a) = GE (OUT (a),M (a)) where M (a) is the set of minimum return values provided as input to the algorithm associated with node a. The definition of M (a) varies based on whether PARENT (a) = *nil* and, if PARENT (a) ≠ *nil*, the value of f (PARENT (a)). If the node is the root of the syntax tree, it has no parent. Otherwise, it may have a parent of type AND, AND NOT, or OR. The remainder of this chapter (after the following example) defines M (a) for each of these four cases accompanied by illustrative examples. These definitions are followed by a summary section which gives the full definition of M (a).

Example 12

This example illustrates the definition of AOUT by giving the value of AOUT for all of the nodes in the syntax tree given in Example 3 in Chapter II. The various values of M which are used in this example are taken from Examples 13 through 16.

AOUT (r) = GE (OUT (r),M (r))

= GE ({ (5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞) }, { (1,1), (5,2), (5,10), (5,12), (7,3), (7,4) })

= { (5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞) }

AOUT (u) = GE (OUT (u),M (u))

= GE ({ (2,3), (5,1), (5,11), (7,2), (∞,∞) }, { (1,1), (3,1), (5,2), (5,12), (7,3) })

= { (2,3), (5,1), (5,11), (7,2), (∞,∞) }

AOUT (v) = GE (OUT (v),M (v))

= GE ({(3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8), (∞,∞)}, {(1,1), (5,1), (5,10), (7,1), (7,4), (∞,1)})

= { (3,2), (5,9), (6,5), (7,3), (8,8), (∞,∞) }

AOUT (w) = GE (OUT (w),M (w))

= GE ({ (1,5), (1,15), (2,3), (3,4), (5,1), (5,11), (7,2), (∞,∞) }, {(1,1), (2,1), (3,1), (4,1), (5,2), (5,12), (7,3) })

= { (1,5), (2,3), (3,4), (5,1), (5,11), (7,2), (∞,∞) }

AOUT (x) = GE (OUT (x),M (x))

= GE ({ (1,7), (3,1), (3,5), (4,2), (6,1), (9,4), (∞,∞) }, { (1,1), (3,1), (5,1), (7,1), (∞,1) })

= { (1,7), (3,1), (6,1), (9,4), (∞,∞) }

AOUT (y) = GE (OUT (y),M (y))

= GE ({ (3,2), (5,9), (6,5), (8,8), (∞,∞) }, { (1,1), (5,1), (5,10), (7,1), (∞,1) })

= { (3,2), (5,9), (6,5), (8,8), (∞,∞) }

AOUT (z) = GE (OUT (z),M (z))

= GE ({ (3,3), (4,7), (7,3), (∞,∞) }, { (1,1), (5,1), (7,4) })

= { (3,3), (7,3), (∞,∞) }

4.2.3 Definition of M for the Root Node

For the root node of the syntax tree, the set of minimum return values is defined by the driver algorithm for the incremental evaluation (see Algorithm 4 in Chapter III). This algorithm uses (1,1) as the initial minimum return value and then uses the value $(d,o+1)$ when the previous output value of the root node is (d,o). So when node a is the root, $M(a) = \{ (1,1) \} \cup \{ (d,o+1) \mid (d,o) \in OUT(a) \}$.

Example 13

This example illustrates the definition of M for the root node of a syntax tree by giving the value of M (r) for the root node r of the syntax tree given in Example 3 in Chapter II.

$$M(r) = \{ (1,1) \} \cup \{ (d,o+1) \mid (d,o) \in OUT(r) \}$$
$$= \{ (1,1), (5,2), (5,10), (5,12), (7,3), (7,4) \}$$

4.2.4 Definition of M for the Children of OR Nodes

For the children of an OR node, the definition of M is based on Algorithm 5 given in Chapter III. Both children are treated equally by the algorithm, so a single definition of M applies to both children. M is composed of two disjoint sets for these children. The first component of M for a child of an OR node is generated by *Step 1* of the initial invocation of Algorithm 5 for the OR node. This component is due to the initial conditions of the incremental algorithms. Namely, l_a is initialized to (-1,-1)

and the first minimum return value passed to the algorithm is always (1,1). This means that $l_m > l_a$ on entry to the first invocation of the algorithm, which in turn means that *Step 1* will pass l_m on to the child of the OR node. Thus, the first component of M is { (1,1) }.

The second component of M is also generated by *Step 1* of Algorithm 5. This step invokes the child of the OR node with l_m as the minimum return value when $l_m > l_a$; however, since the invocation replaces l_a with a new value, only the first $l_m > l_a$ will be used as a minimum return value for the child a. In other words, if f (PARENT(a)) = OR for some node a, then for each $l \in$ OUT(a) only the smallest element of M(PARENT(a)) that is larger than l is also an element of M(a). So the set G (M (PARENT (a)),OUT (a)) is the second component of M for the children of an OR node.

Example 14

This example illustrates the definition of M for nodes which are children of OR nodes by giving the value of M (y) and M(z) where y and z are nodes in the syntax tree given in Example 3 in Chapter II.

M (y) = { (1,1) } ∪ G (M (v),OUT (y))
= { (1,1) } ∪ G ({(1,1), (5,1), (5,10), (7,4), (∞,1) }, { (5,2), (5,9), (6,5), (8,8), (∞,∞) })
= { (1,1), (5,1), (5,10), (7,1), (∞,1) }

M (z) = { (1,1) } ∪ G (M (v),OUT (z))

$= \{ (1,1) \} \cup G (\{(1,1), (5,1), (5,10), (7,1), (7,4), (\infty,1)\}, \{(3,3), (4,7), (7,3), (\infty,\infty) \})$

$= \{ (1,1), (5,1), (7,4) \}$

4.2.5 Definition of M for the Children of AND NOT Nodes

For the children of an AND NOT node, the definition of M is based on Algorithm 6 given in Chapter III. This algorithm treats the left and right children of the node to which it is associated differently, resulting in different definitions of M for the two children. For the left child M is composed of two disjoint sets, and for the right child it is composed of three disjoint sets.

Algorithm 6 always returns a value l_a. The minimum return value l_m is always greater than the return value l_r of the previous invocation, or greater than (-1,-1) when there is no previous invocation, thus $l_m > l_a$ at the beginning of each invocation of Algorithm 6. Therefore, l_m is always passed to the left child by *Step 1* of Algorithm 6. So, the first component of M for the left child a of an AND NOT node is M (PARENT (a)).

The second component of M for the left child of an AND NOT node is generated by *Step 5* of Algorithm 6. This step invokes the left child when $l_a = (d,o)$ and $l_b = (d,o')$ for some document number d. The minimum return value used in this invocation is $(d+1,1)$ so that l_a will be replaced by the smallest value in OUT (a) which has a larger document number. *Step 5* is executed in a loop which begins with $l_a > l_m$ and $l_b > l_m$ and ends when $l_a \in$ OUT (PARENT (a)); however, *Step 5* is not

executed if $l_a = (\infty,\infty)$. Taking this all into consideration, one can see that the following set is the second component of M for the left child of an AND NOT node: $\{ (d+1,1) \mid d \neq \infty$ and for some $o, o' \in O_d$, $(d,o) \in$ LV (OUT (a), G (OUT (a), M (PARENT (a))), OUT (PARENT (a))) and $(d,o') \in$ LV (OUT (SIB (a)), G (OUT (SIB(a)), M (PARENT (a))), OUT (PARENT (a))) $\}$.

The first component of M for a right child of an AND NOT node is { (1,1) } due to the initial conditions for the incremental algorithms. This component and the reasoning for it are the same as the first component of M for the children of an OR node (see Section 4.2.4).

The second component of M for a right child b of an AND NOT node is generated by *Step 1* of Algorithm 6. On entry to Algorithm 6 (except for the first invocation that is part of the initial conditions discussed above), $l_b \in$ GE (OUT (b), $\{l_p\}$) where l_p is the return value from the previous invocation of the algorithm (l_r at the end of that invocation). Also, $l_m \in$ G (M (PARENT (b)), $\{l_p\}$). If $l_m > l_b$, then *Step 1* will invoke child b with l_m as the minimum return value. So, the second component of M is $\{ l_m \mid l_m \in$ G (M (PARENT (b)), $\{l_p\}$), for some $l_p \in$ OUT (PARENT (b)) and $l_m > l_b$ for $l_b \in$ G (OUT (b), $\{l_p\}$) $\}$.

The third component of M for the right child of an AND NOT node is generated by *Step 4* of Algorithm 6. This step invokes the right child with $(d_a,1)$ as the minimum return value when $d_a > d_b$, where $l_a = (d_a,o_a)$ and $l_b = (d_b,o_b)$. Note that only the first l_a which meets this condition will be used to generate a minimum

return value (d_a,1). *Step 4* is executed in a loop which begins with $l_a \in$ GE (OUT(a), $\{l_m\}$) and $l_b \in$ GE (OUT (b), $\{l_m\}$), and ends with $l_a \in$ GE(OUT (PARENT (a)), $\{l_m\}$). This can be restated formally by the following definition of the set which is the third component of M for the right child b of an AND NOT node: $\{ (d,1) \mid d \neq \infty$, and for some $o \in O_d$, $(d,o) \in$ OUT (SIB (b)) and $l_m \leq (d,o) \leq l_p$ for some $l_m \in$ M (PARENT (b)), and $l_p \in$ G (OUT (PARENT (b)), $\{l_m\}$), and $d > d'$ for some $(d',o') \in$ OUT (b) such that $(d',o') \geq l_m$ and $\nexists$ $(d'',o'') \in$ OUT (SIB (b)) such that $d'' > d'$ and $(d'',o'') < (d,o)$ $\}$.

Example 15

This example illustrates the definition of M for nodes which have AND NOT nodes as parents by giving the value of M for the nodes w and x in the syntax tree given in Example 3 in Chapter II.

M (w) = M (u) $\cup$ $\{ (d+1,1) \mid d \neq \infty$ and for some $o, o' \in O_d$, $(d,o) \in$ LV (OUT (w), G (OUT(w), M(u)), OUT(u)) and $(d,o') \in$ LV (OUT (x), G (OUT (x)), M(u))), OUT(u)) $\}$

= { (1,1), (3,1), (5,2), (5,12), (7,3) } $\cup$ { (2,1), (4,1) }

= { (1,1), (2,1), (3,1), (4,1), (5,2), (5,12), (7,3) }

M (x) = { (1,1) } $\cup$ $\{ l_m \mid l_m \in$ G (M (u), $\{l_p\}$) for some $l_p \in$ OUT (u) and $l_m > l_a$ for $l_a \in$ G (OUT (x), $\{l_p\}$) $\}$ $\cup$ $\{ (d,1) \mid d \neq \infty$ and for some $o \in O_d$, $(d,o) \in$ OUT (w) and $l_m \leq (d,o) \leq l_p$ for some $l_m \in$ M (u) and $l_p \in$ G (OUT (u), $\{l_m\}$) and $d > d'$ for some $(d',o') \in$ OUT (x) such that $(d',o') \geq l_m$ and $\nexists (d'',o'') \in$ OUT (w) such that $d'' > d'$ and $(d'',o'') < (d,o)$ $\}$

= { (1,1) } $\cup$ $\emptyset$ $\cup$ { (1,1), (3,1), (5,1), (7,1) }

$= \{ (1,1), (3,1), (5,1), (7,1) \}$

4.2.6 Definition of M for the Children of AND Nodes

For the children of an AND node, the definition of M is based on Algorithm 7 given in Chapter III. Both children are treated equally by Algorithm 7, so the same definition applies to both nodes. M is composed of three disjoint sets for these children. The first component of M is { (1,1) } due to the initial conditions for the incremental algorithms. This component and the reasoning for it are the same as the first component of M for the children of an OR node (see Section 4.2.4).

The second component of M is generated by *Step 1* of Algorithm 7. On entry to Algorithm 7 (except for the first invocation that is part of the initial conditions discussed above), $l_a \in$ GE (OUT (a), $\{l_p\}$) where l_p is the return value from the previous invocation of the algorithm (l_r at the end of that invocation) and a is either of the children of the AND node. Also, $l_m \in$ G (M (PARENT (a)), $\{l_p\}$). If $l_m > l_a$, then *Step 1* will invoke child a with l_m as the minimum return value. So, the second component of M is $\{ l_m \mid l_m \in$ G (M (PARENT (a)), $\{l_p\}$), for some $l_p \in$ OUT (PARENT (a)) and $l_m > l_a$ for $l_a \in$ G (OUT (a), $\{l_p\}$) $\}$.

The third component of M for a child of an AND node is generated by either *Step 5* or *Step 6* of Algorithm 7 depending on whether the child is the left child or right child, respectively. The two children are treated identically, so, without loss of generality, we can consider only node a, the left child of an AND node. In this case,

Step 5 invokes the algorithm for node a any time $d_a < d_b$, or equivalently $l_a < l_b$, since *Step 4* catches the case where $d_a = d_b$. Note that only the first value of l_b which is larger than l_a will cause *Step 5* to invoke the node a. In other words, only $l_b \in$ G (OUT (SIB (a)), { l_a}) is used or, in general, only the set G (OUT (SIB (a)), OUT (a)) can provide a location (d,o) which will be used to provide a minimum return value of (d,1) to node a. *Step 5* is executed in a loop which begins with $l_a \in$ GE (OUT (a), {l_m}) and $l_b \in$ GE (OUT (SIB (a)), {l_m}) for some $l_m \in$ M (PARENT (a)), and ends with $l_a \in$ OUT(PARENT (a)) or $l_b \in$ OUT (PARENT(a)). The function LV selects values which are between an upper bound and a lower bound from one set, so the third component of M is the set { (d,1) | for some $o \in$ O_d, (d,o) $\in$ LV (G (OUT (SIB (a)), OUT (a)), GE(OUT(a), M (PARENT (a))), OUT (PARENT (a))) }.

Example 16

This example illustrates the definition of M for the children of AND nodes by giving the value of M for the nodes u and v in the syntax tree given in Example 3 in Chapter II.

M (u) = { (1,1) } $\cup$ { l | $l \in$ G (M (r), { l_p}) for some $l_p \in$ OUT (r) and $l > l_a$ for $l_a \in$ GE (OUT (u), {l_p}) } $\cup$ { (d,1) | for some $o \in O_d$, (d,o) $\in$ LV (G (OUT (v), OUT (u)), GE (OUT (u), M (r)), OUT (r)) }

= { (1,1 } $\cup$ { l | $l \in$ G ({ (1,1), (5,2), (5,1), (5,12), (7,3), (7,4) }, { l_p}) for some $l_p \in$ { (5,1), (5,9), (5,11), (7,2), (7,3), (8,8), (∞,∞) } and $l > l_a$ for $l_a \in$

GE({(2,3), (5,1), (5,11), (7,2), (∞,∞) }, {l_p}) } ∪ { (d,1) | for some $o \in O_d$, $(d,o) \in$ LV (G ({ (3,2), (3,3), (4,7), (5,9), (6,5), (7,3), (8,8), (∞,∞) }, { (2,3), (5,1), (5,11), (7,2), (∞,∞) }), GE ({ (2,3), (5,1), (5,11), (7,2), (∞,∞) }, { (1,1), (5,2), (5,10), (5,12), (7,3), (7,4) }), { (5,1), (5,9), (5,11), (7,2), (7,3), (∞,∞) }) }

= { (1,1) } ∪ { (5,2), (5,12), (7,3) ∪ { (3,1) }

= { (1,1), (3,1), (5,2), (5,12), (7,3) }

M (v) = { (1,1) } ∪ { l | $l \in$ G (M (r), { l_p }) for some $l_p \in$ OUT (r) and $l > l_a$ for $l_a \in$ GE (OUT (v), {l_p}) } ∪ { (d,1) | for some $o \in O_d$, $(d,o) \in$ LV (G (OUT (u), OUT (v)), GE (OUT (v), M (r)), OUT (r)) }

= { (1,1) } ∪ { (5,10), (7,4) } ∪ { (5,1), (7,1), (∞,1) }

= { (1,1), (5,1), (5,10), (7,1), (7,4), (∞,1) }

4.2.7 Full Definition of M

The preceding definitions are summarized in the following complete definition of M. From Section 3.2, L′ is the domain of the minimum return values. Let ST = (BT, r, f), BT = (A, R) be a syntax tree and $a \in$ A, then M : A → $2^{L'}$, where $2^{L'}$ is the power set of L′, is defined as

(i) if $a = r$,
M (a) = { (1,1) } ∪ { (d,o + 1) | (d,o) ∈ OUT (a) };

(ii) if f (PARENT (a)) = OR,
M (a) = { (1,1) } ∪ G (M (PARENT (a)), OUT (a));

(iii) if f (PARENT (a)) = AND NOT and LEFT (PARENT (a)) = a,

$$M(a) = M(\text{PARENT}(a)) \cup \{ (d+1,1) \mid d \neq \infty \text{ and for some } o, o' \in O_d,$$
$$(d,o) \in LV(\text{OUT}(a), G(\text{OUT}(a), M(\text{PARENT}(a))),$$
$$\text{OUT}(\text{PARENT}(a))) \text{ and } (d,o') \in LV(\text{OUT}(\text{SIB}((a)),$$
$$G(\text{OUT}(\text{SIB}(a)), M(\text{PARENT}(a))), \text{OUT}(\text{PARENT}(a))) \}; \text{ and}$$

(iv) if $f(\text{PARENT}(a)) = \text{AND NOT}$ and $\text{RIGHT}(\text{PARENT}(a)) = a$,
$$M(a) = \{ (1,1) \} \cup \{ l_m \mid l_m \in G(M(\text{PARENT}(a)), \{l_p\}) \text{ for some}$$
$$l_p \in \text{OUT}(\text{PARENT}(a)) \text{ and } l_m > l_a \text{ for } l_a \in G(\text{OUT}(a), \{l_p\}) \} \cup$$
$$\{ (d,1) \mid d \neq \infty \text{ and for some } o \in O_d, (d,o) \in \text{OUT}(\text{SIB}(b)) \text{ and}$$
$$l_m \leq (d,o) \leq l_p \text{ for some } l_m \in M(\text{PARENT}(b)) \text{ and } l_p \in$$
$$G(\text{OUT}(\text{PARENT}(b)), \{l_m\}) \text{ and } d > d' \text{ for some } (d',o') \in \text{OUT}(b)$$
$$\text{such that } (d',o') \geq l_m \text{ and } \nexists (d'',o'') \in \text{OUT}(\text{SIB}(b)) \text{ such that } d'' > d'$$
$$\text{and } (d'',o'') < (d,o) \};$$

(v) if $f(\text{PARENT}(a)) = \text{AND}$,
$$M(a) = \{ (1,1) \} \cup \{ l_m \mid l_m \in G(M(\text{PARENT}(a)), \{l_p\}) \text{ for some}$$
$$l_p \in \text{OUT}(\text{PARENT}(a)) \text{ and } l_m > l_a \text{ for } l_a \in G(\text{OUT}(a), \{l_p\}) \} \cup$$
$$\{ (d,1) \mid \text{for some } o \in O_d, (d,o) \in LV(G(\text{OUT}(\text{SIB}(a)), \text{OUT}(a)),$$
$$GE(\text{OUT}(a), M(\text{PARENT}(a))), \text{OUT}(\text{PARENT}(a))) \}.$$

This chapter formalized the cosequential query evaluation method and the incremental query evaluation method. Namely, the outputs of each of the algorithms in each of the methods were formally defined for a given syntax tree representing a query of a full-text database. For the incremental method, this definition required a rather extensive definition of the set of minimum return values for a given query. The next chapter uses the formalisms provided in this chapter and the algorithms defined in Chapter III to compare the cosequential query evaluation method to the incremental query evaluation method.

CHAPTER V

COMPARISON OF THE EVALUATION METHODS

This chapter compares the cosequential and incremental query evaluation methods by analyzing their space and time complexities.

5.1 Space Complexity

The query evaluation algorithms require a constant amount of space for code, instance variables, stack frames, etc. This space is dependent upon the computing environment in which the algorithms are executed, but it is relatively minor in quantity and it is asymptotically insignificant.

The major dynamic space utilization is in the input and output document lists. Both query evaluation methods take the same input document lists at the level of the leaf nodes in the syntax tree and produce the same output document lists from the root node. This leaves only the internally generated document lists for comparing the space utilization of the two methods.

Let ST = (BT, r, f), BT = (A, R) be a syntax tree. From Section 4.1, the document list generated from a cosequential algorithm for syntax tree node a is

OUT (a). Thus $S = \sum_{a \in A} |OUT(a)|$ is the total number of document list elements produced during the cosequential evaluation of the syntax tree ST. As a result, an upper bound for the space complexity of the cosequential evaluation method is O(S - |OUT(r)|) in terms of the number of document list elements stored. This approximation assumes that none of the space used to store document lists is reused during the evaluation process.

A more accurate approximation for the space complexity requires that the reuse of space be accounted for. Consider a path $p = (a_1, a_2, ..., a_n)$ in the syntax tree ST with $a_1 = r$ and $n \geq 1$. Such a path represents a state of the query evaluation process. In this state nodes a_1 through a_{n-1} are pending and node a_n is active (i.e., it has received all its required input and is about to process that input). The cosequential evaluation process performs a postorder traversal of the syntax tree (see Chapter III). We can assume without loss of generality that the traversal proceeds in the usual left-to-right manner. Thus, for each node a_i, the output of its left child on the path is pending (i.e., waiting to be consumed) unless that left child is the next node in the path, viz. a_{i+1}. In addition, the output of the right child of the last node in the path, a_n, is pending if a_n is not a leaf node. The sum of the sizes of all pending output document lists is the space utilization for the path p. Thus, the following statements express the space utilization s for the path p.

$$x_i = \begin{cases} |OUT(LEFT(a_i))|, & \text{if LEFT } (a_i) \neq a_{i+1} \text{ and } i < n \\ 0, & \text{if LEFT } (a_i) = a_{i+1} \text{ and } i < n \end{cases}$$

$$s' = \begin{cases} |\text{OUT}(\text{LEFT}(a_n))| + |\text{OUT}(\text{RIGHT}(a_n))|, & \text{if } f(a_n) \neq \text{LU} \\ 0, & \text{if } f(a_n) = \text{LU} \end{cases}$$

$$s = s' + \sum_{1 \leq i < n} x_i$$

Now, let P = $\{p_1, p_2, ..., p_m\}$ be the set of all paths from the root r to the leaves of the syntax tree ST. Let S = max $\{s_1, s_2, ..., s_m\}$, where s_i is the space utilization for path p_i, be the space utilization for the syntax tree ST. The space utilization for the cosequential evaluation of the syntax tree is then O(S).

Each of the incremental query evaluation algorithms produces a single output value; hence the space complexity for the incremental evaluation method is O(|A|) for the syntax tree ST.

Therefore, for most syntax trees the space utilization of the incremental query evaluation method is less than that of the cosequential query evaluation method. Exceptions to this are syntax trees consisting of a single node and the case where all input document lists consist of only the end-of-list marker (∞,∞). In the former case, there are no internally generated document lists, and in the latter case, each internally generated document list contains exactly one value which equals the space utilization for the incremental evaluation algorithms.

5.2 Time Complexity

Comparing the time complexities of the cosequential and incremental query evaluation algorithms is more difficult than comparing their space complexities and the results are not as clear.

One of the factors involved in time complexity analysis is input and output processing. Some amount of input processing is required to retrieve the document lists provided as input to the leaf nodes of the syntax tree because these document lists will probably be retrieved from an inverted file. This amount of time is ignored here because the inverted file is not a topic of this thesis and can be implemented in various manners, thus making general statements about its performance impossible.

The cosequential evaluation algorithms require additional input/output processing. Each cosequential algorithm produces a document list containing all database locations in the input document lists meeting the criteria of the search operator implemented by that algorithm; hence, the size of the output document list is based on the sizes of the input document lists. When accessing a large database, the document lists input to the query evaluation process can be quite large because each list typically represents all occurrences of a particular word. Hence, the document lists produced by the cosequential algorithms can be quite large. In a typical computing environment, these large document lists will have to be stored on a mass storage device until they are consumed in a subsequent step of the evaluation. Thus, the document lists will have to be written to a mass storage device as they are

produced and read from a mass storage device as they are consumed. Therefore, if ST = (BT, r, f), BT = (A, R) is a syntax tree, then the time for input/output processing in the cosequential evaluation method would be $O\left(\sum_{a \in A} |OUT(a)|\right)$.

The incremental query evaluation algorithms produce a single value per invocation, so there is no need to store these values on mass storage devices until they become part of the final document list. Hence, the input/output processing time for the incremental evaluation method is $O(|OUT(r)|)$ for storing the values produced by the root of the syntax tree. So, $T_1 = O\left(\sum_{a \in A} |OUT(a)|\right) - O(|OUT(r)|)$ or

$O\left(\sum_{\substack{a \in A, \\ a \neq r}} |OUT(a)|\right)$ is the difference between the input/output processing times for the incremental and cosequential evaluation methods.

The other main factor in the difference in the time complexities of the cosequential and incremental evaluation methods is the time spent executing the instructions in the algorithms. Because this time is highly sensitive to the computing environment in which the algorithms are run and because the exact instructions necessary are also dependent on that environment, the comparison of processing times that follows is based on the number of steps used by the algorithms.† For the

---

† As with the input/output processing time comparison, the step comparison does not include the time required to access the inverted file to obtain the document lists provided as input to the leaf nodes of the syntax tree.

purposes of this analysis, a step is defined to be one function invocation (including INPUT and OUTPUT which read or write a value), one function return statement, one comparison with the accompanying branches (including a three-way branch), or one assignment.

Let ST = (BT, r, f), BT = (A, R) be a syntax tree with $c \in A$. In addition, if $f(c) \neq$ LU, let $a, b \in A$ with LEFT(c) = a and RIGHT(c) = b. Let d_c be the number of different documents in OUT (c) and d_i be the number of different documents in AOUT (c). TABLE IX shows the value of E_c and E_i, the approximate number of

TABLE IX

APPROXIMATE NUMBER OF STEPS FOR THE TWO QUERY EVALUATION METHODS

$f(c)$	E_c	E_i
AND	$2(\|OUT(a)\| + \|OUT(b)\| + \|OUT(c)\| + d_c)$	$2(\|AOUT(a)\| + \|AOUT(b)\| + d_i) + 4\|AOUT(c)\|$
OR	$3\|OUT(c)\|$	$5\|AOUT(c)\|$
AND NOT	$2(\|OUT(a)\| + \|OUT(b)\|) + \|OUT(c)\|$	$2(\|AOUT(a)\| + \|AOUT(b)\|) + 3\|AOUT(c)\|$
LU	$3\|OUT(c)\|$	$2\|AOUT(c)\|$

$f(c)$ is the node type.
E_c is the number of steps for the cosequential evaluation method.
E_i is the number of steps for the incremental evaluation method.

steps required to evaluate node c using the cosequential evaluation method and the incremental evaluation method, respectively, for each possible value of $f(c)$.

For the sake of simplicity, some assumptions have been made in calculating these approximations. First, the likelihood of a database location appearing in both document lists input to an algorithm is assumed to be negligible. This assumption forces the approximations to overestimate E_c and E_i slightly because the approximations assume that each pass through the main loop of an algorithm either rejects or accepts one database location. When the two current database locations are equal, one pass through the loop either rejects or accepts both database locations.

A second assumption is that an optimization on the placement of the test for the end-of-list marker has been removed from the AND and AND NOT algorithms in both evaluation methods. This optimization allows the AND algorithms in both evaluation methods to terminate immediately when the end-of-list marker is found in either input list, and it also allows the AND NOT algorithms in both evaluation methods to terminate when the end-of-list marker is found in the input list from the left child. This optimization has a mixed effect on performance because it moves a test from a low-use branch into the main flow of control, but it can save a large number of comparisons when one input list is significantly shorter than the other (especially in the incremental algorithms where the optimization augments the performance gains resulting from the use of the minimum return value). The assumption that this optimization has been removed eliminates the need for an addi-

tional term (and complications that are entailed by that) to account for the input values which are never accessed.

A third assumption is that the minimum return value causes only one value to be input at the beginning of each invocation of one of the incremental evaluation algorithms. In some cases when the minimum return value has been determined by an ancestor AND or AND NOT algorithm (rather than by Algorithm 4, the incremental driver algorithm), the minimum return value will increase so much that it would be larger than both of the current input values. When this happens, a new value must be input from both children rather than just one or the other. This assumption causes the approximations to overestimate in the same manner as the first assumption. One additional step is executed, but an entire sequence of steps which normally determines the fate of one input value now determines the fate of two input values.

Notice that the entries in TABLE IX are very similar for a given value of $f(c)$ with only two differences: some of the constants for the incremental algorithms are greater, and the terms for all of the incremental algorithms are proportional to the AOUT sets while the cosequential algorithms are proportional to the OUT sets.

The difference in the constants for the cases where $f(c) \neq$ LU is the cost of the two extra comparisons required to implement the minimum return value used in the incremental evaluation method. The difference between the use of the OUT and AOUT sets represents the potential benefit of those comparisons due to smaller input and output document lists.

The algorithms for the LU nodes of the syntax tree have not been discussed previously because of their simplicitly and lack of impact on prior discussions. They consist of an invocation of the inverted file followed by an output of the value returned from that invocation (or a return of that value in the case of the incremental LU algorithm). In addition, the cosequential version requires a comparison for each value to determine when the end of the input document list has been reached. The incremental version simply will not be invoked again once it has returned the end-of-list marker. The foregoing discussion accounts for the expressions in TABLE IX where $f(c)$ = LU.

Thus, we have $T_2 = \sum_{a \in A} E_c(a) - (3|AOUT(r)| + \sum_{a \in A} E_i(a))$ as the difference between the number of steps required to execute the cosequential and incremental evaluation algorithms. The term $3|AOUT(r)|$ is due to the incremental driver algorithm (Algorithm 4).

Hence, the total difference between the time complexities of the cosequential and incremental evaluation methods would be $T = \alpha T_1 + \beta T_2$, where α and β are scale factors to convert the number of execution steps and input/output processing values to an equivalent time measurement. The values of α and β are dependent on the computing environment and the implementation language.

The value of T can be computed given the values of α and β, the syntax tree ST and the input document list for each leaf node of ST. However, this value would be relevant only to that particular query. To make a general statement about the rela-

tive performance of the evaluation methods, it is necessary to have a model of the "typical" use of a "typical" full-text database.

The general model would include three components. First, the document lists input to the query evaluation process must be represented. This representation must characterize the size of input document lists in terms of the number of database locations included as well as the number of documents included. It must also characterize the relationships among the elements of various document lists. These characterizations are necessary for an effective representation of the size of the output of the evaluation algorithms relative to the input and the number of steps required to produce that output.

The Zipfian distribution is commonly used to model the distribution of words within a database ([5] for example). These models can express the relationships among the occurrences of words within a database; however, they must be combined with a model of the words chosen in queries before they can model the relationships among the document lists input to a query evaluation.

A second necessary component of the model is a representation of the syntax tree. The number and placement of the query operators within a syntax tree must be characterized to allow the affect of the minimum return value to determined. [4] is an example which models some of the necessary aspects for a particular online system. This model does not include any representation of the relationships among the

operators within a query. Also, because it is based on data from a particular online system, it could be difficult to generalize.

The final component of the model is a representation of the computing environment. This is necessary to determine the relative costs of the input/output and step execution as well as a realistic description of the actual steps required to evaluate a query. This is probably the easiest component of the model to define. An existing computing environment which is capable of handling the large amount of storage and processing necessary for evaluating queries in very large databases could be identified and the characteristics of that environment used.

The definition of the model described is beyond the scope of this thesis because of the large amounts of history data required. A conclusive comparison of the incremental and cosequential evaluation methods requires such a model.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This thesis contains the definitions of two methods of evaluating queries in very large full-text databases. The two methods are called cosequential evaluation and incremental evaluation. The cosequential method is the more commonly used method of the two. It evaluates query elements in pairs until a single document list results. The incremental method is a method which evaluates all query elements in an incremental fashion. Algorithms were presented to evaluate the AND, OR, and AND NOT query operators using both evaluation methods.

A formalism was defined for the discussion of the query evaluation methods. This formalism includes representations of the syntax tree resulting from the parsing of the query (ST), the output of each cosequential algorithm (OUT), the output of each incremental algorithm (AOUT), and the sets of minimum return values (M) used in the incremental evaluation method.

An attempt was made to compare the evaluation methods asymptotically. This comparison showed that the cosequential evaluation method can require significantly larger amounts of disk storage and input/output processing time than the incremental evaluation method. The comparison also showed that the number of

steps executed during the query evaluation was similar for both evaluation methods. The incremental evaluation method pays a price for the use of the minimum return values, but it also benefits from the minimum return values.

The comparison was unable to determine whether the cost of the minimum return values outweighs the benefits because an adequate model of the query evaluation system (including the computing environment, the full-text database, and the queries) was not available.

One area of future work is the definition of a theoretical model of the query evaluation system that includes all of the described components. With that model in hand, the complexity analysis begun here could be completed. Another area of future work is the empirical evaluation of the two query evaluation methods. This may be easier than the definition of a full model of the query evaluation system, but it does require history data describing a typical database and the typical queries used to retrieve information from that database.

Another area of future work is finding query evaluation methods that improve on the cosequential and incremental query evaluation methods presented here. One possibility is a combination of the cosequential and incremental query evaluation methods. It might be possible to examine a given query and use the most efficient method for that query.

The incremental query evaluation method reduces access to secondary storage by eliminating large intermediate results and by using the query in a holistic

manner to reduce the number of database locations which must be examined for each lexical unit in the query. It would interesting to compare these reductions to the reductions achieved by methods based on the optimizations in relational database systems, methods that are expressly designed to optimize the secondary storage access. (e.g., [7], [14], and [17]).

REFERENCES

[1] —, *RESEARCH Retrieval Version 2.1 – User's Guide*, TMS, Inc., 110 W. Third, Stillwater, OK, 74074, (1987).

[2] —, *RESEARCH Database Preparation 2.1 – User's Guide*, TMS, Inc., 110 W. Third, Stillwater, OK, 74074, (1987).

[3] Alfred Aho and Jeffery D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*, Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1972.

[4] Michael D. Cooper, "Usage Patterns of an Online Search System," *Journal of the American Society for Information Science*, Vol. 34, No. 5 (May 1983), pp. 343-349.

[5] Jane Fedorowicz, "Database Performance Evaluation in an Indexed File Environment," *ACM Transactions on Database Systems*, Vol. 12, No. 1 (March 1987), pp. 85-110.

[6] Michael J. Folk and Bill Zoellick, *File Structures: A Conceptual Toolkit*, Addison-Wesley Publishing Company, (1987).

[7] Farshad Fotouhi and Sakti Pramanik, "Optimal Secondary Storage Access Sequence for Performing Relational Join," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 3 (September 1989), pp. 318-328.

[8] Johann Christoph Freytag, "A Rule-Based View of Query Optimization," *ACM SIGMOD Record*, Vol. 16, No. 3 (December 1987), pp. 173-180.

[9] Jeffrey A. Hoffer and Antonio Kovacevic, "Optimal Performance of Inverted Files," *Operations Research*, Vol. 30, No. 2 (March/April 1982), pp. 336-354.

[10] Yannis E. Ioannidis and Eugene Wong, "Query Optimization by Simulated Annealing," *ACM SIGMOD Record*, Vol. 16, No. 3 (December 1987), pp. 9-22.

[11] Matthias Jarke and Jürgen Koch, "Query Optimization in Database Systems," *Computing Surveys*, Vol. 16, No. 2 (June 1984), pp. 111-152.

[12] Whay C. Lee and Edward A. Fox, "Experimental Comparison of Schemes for Interpreting Boolean Queries," Technical Report TR 88-27, Department of Computer Science, Virginia Polytechnic Institute and State University, 1988, 117 pages.

[13] Jane W. S. Liu, "Algorithms for Parsing Search Queries in Systems with Inverted File Organization," *ACM Transactions on Database Systems*, Vol. 1, No. 4 (December 1976), pp. 299-316.

[14] Anne Putkonen, "The Order of Merging Operations for Queries in Inverted File Systems," *International Journal of Computer and Information Sciences*, Vol. 9, No. 5 (October 1980), pp. 351-369.

[15] Gerard Salton, Edward A. Fox, and Harry Wu, "Extended Boolean Information Retrieval," *Communications of the ACM*, Vol. 26, No. 12 (December 1983), pp. 1022-1036.

[16] Sreekumar T. Shenoy and Zehra Meral Ozsoyoglu, "Design and Implementation of a Semantic Query Optimizer," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 3 (September 1989), pp. 344-361.

[17] S. B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Transactions on Database Systems*, Vol. 4, No. 2 (June 1979), pp. 133-155.

[18] S. B. Yao, "Approximating Block Accesses in Database Organizations," *Communications of the ACM*, Vol. 20, No. 4 (April 1977), pp. 260-261.

VITA γ

Rodney Lee Barnett

Candidate for the Degree of

Master of Science

Thesis: FORMALIZATION AND COMPARISON OF TWO QUERY EVALUATION METHODS IN VERY LARGE FULL-TEXT DATABASES

Major Field: Computer Science

Biographical:

Personal Data: Born in Edmond, Oklahoma, February 20, 1963, son of Cliff and Betty Barnett.

Education: Graduated from Edmond Memorial High School, Edmond, Oklahoma, in May 1981; attended Southern Methodist University, Dallas, Texas during the 1981-82 academic year; received Bachelor of Science Degree in Computing and Information Sciences and Mathematics at Oklahoma State University in May 1986; completed requirements for the Master of Science degree in Computer Science at Oklahoma State University in December 1991.

Professional Experience: Part-time computer programmer, TMS, Inc., Stillwater, OK, January 1985 to June 1986; full-time software engineer, TMS, Inc., July 1986 to March 1991; senior software engineer, Teltech, Minneapolis, MN, March 1991 to present.