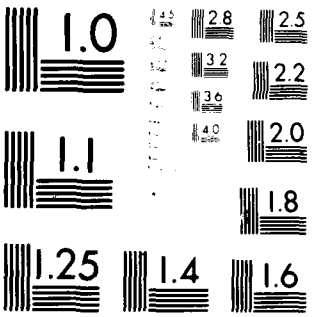
AD-A110 139 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 5/2
PREDICATE-ORIENTED DATABASE SEARCH ALGORITHMS.(U)
MAY 78 D E WILLARD N00014-76-C-0914
UNCLASSIFIED TR-20-78 NL

1 OF 3
AD
A110139



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

LEVEL II

(12)

PREDICATE-ORIENTED DATABASE SEARCH ALGORITHMS*

by

Dan E. Willard

TR-20-78

(12)

*This research was supported in part by the Office of Naval Research under contract N00014-76-C-0914.

This document [illegible]
for publi[illegible]
distribution [illegible]

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A110139	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Predicate-Oriented Database Search Algorithms	5. TYPE OF REPORT & PERIOD COVERED technical report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Willard, Dan E.	8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0914
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University, Aiken Computation Laboratory, Cambridge, Massachusetts 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, 800 North Quincy Street, Arlington, Virginia 22217	12. REPORT DATE May 1978
	13. NUMBER OF PAGES 384 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

multidimensional searching
relational databases
B-tree
super-B-tree
low-level SUM and COUNT algorithms
low-level FIND algorithms
E-7 algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Let $e(x,y)$ denote an arbitrarily complex (nonatomic) predicate, and let $e(\bar{x},y)$ denote the special form that this prediate will assume when the initial multi-component variable of x is replaced by the fixed tuple of $\bar{x}$. This thesis will study the question of how the set of y-records that satisfy the condition of $e(\bar{x},y)$ can be efficiently retrieved. Such retrievals will be examined in the context of a "dynamic" environment where new records are being continually inserted, deleted, and updated. This thesis will show that

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

every conceivable predicate of $e(x,y)$ can be assigned a database and a number of $D(e)$ such that

i) For any element of $\bar{x}$, the set of y-records that satisfy $e(x,y)$ can be retrieved in $O(\log^{D(e)} N)$ time (from the initial N-membered database).

ii) The relevant database will be sufficiently flexible so that the user can insert, delete, or modify any of its y-records in $O(\log^{D(e)} N)$ time.

The above results will be extremely significant because the number of $D(e)$ will usually assume small values such as zero, one, or two.

The principal application of this thesis will be in the area of automatic programming. The purpose of that branch of computer science has been to discover how automatic algorithms can be developed which do much of the programming that has traditionally been assigned to human beings.

Such automatic algorithms have been advocated by many computer scientists (Codd-70, Boehm-72, Date-77) because these procedures would dramatically reduce the cost of writing computer programs. The combined work of the cited authors have shown that,

i) the cost of developing computer software may greatly exceed hardware costs in the 1980's (Boehm has estimated that computer programmer labor costs will constitute 90 percent of all the Air Force's 1985 computer-related expenditures),

ii) and that the health of the computer industry requires lower software development costs (even if this is done in the context of a trade-off that modestly increases the hardware costs).

The importance of automatic database search algorithms was further confirmed in a recent panel discussion. ~~(Codd-75).~~ The members of that panel concluded that such automatic search algorithms would be extremely useful if these algorithms could be made to be moderately efficient. This thesis will lay the foundations of the theory that should be used in the development of authomatic predicate searching algorithms.

ACKNOWLEDGEMENT

There are several people who I want to thank with regards to this thesis. First, I thank my thesis adviser, Professor Ugo Gagliardi, because he suggested that I read the previous relational database literature and consider how such a system should be implemented. I am especially grateful to Professor Gagliardi because he realized that relational database optimization was a potentially important topic that had not been widely researched.

I thank all the readers for having examined this thesis. In addition to Professor Gagliardi, these include Professors Harry Lewis and Gerald Sacks. I also thank Professor Garrett Birkhoff who examined this thesis but was unable to be a reader because he was out of town.

Finally I want to express my appreciation to Barbara Moody and Mary Bosco for their excellent typing.

DTIC
COPY
INSPECTED
1

TABLE OF CONTENTS

		Page
CHAPTER 1: INTRODUCTION		
1.1	Abstract	1
1.2	Objectives of this Thesis	3
1.3	An Intuitive Explanation of the Main Algorithms	17
1.4	Organization of the Thesis	19
CHAPTER 2: THE RELATIONAL DATABASE LITERATURE		
2.1	General Description of Previous Research	24
2.2	A More Detailed Description of Previous Research	28
CHAPTER 3: SUPER-B-TREES		
3.1	Chapter Overview	40
3.2	An Examination of B-tree Theory	42
3.3	An Intuitive Description of Super-B-trees	74
3.4	The Mainline of the Alg(α, β, K, J) Procedure	80
3.5	The Proof of the Correctness of the Alg(α, β, K, J) Procedure	103
3.6	An Analysis of the Runtime of Alg(α, β, K, J)	121
CHAPTER 4: THE LOW-LEVEL SUM AND COUNT ALGORITHMS		
4.1	Chapter Overview	150
4.2	Hashing	153
4.3	The SUM-1 Algorithm	163
4.4	The SUM-2 and SUM-3 Algorithms	170

Predicate-Oriented Database Search Algorithms

A thesis presented

by

Dan E. Willard

to

The Department of Mathematics

In partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Mathematics

Harvard University

Cambridge, Massachusetts

May, 1978

Copyright 1978 by Dan E. Willard

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Dan E. Willard.

Page

CHAPTER 5: THE LOW-LEVEL FIND ALGORITHMS

5.1 Chapter Overview 185

5.2 An Intuitive Description of the FIND-1 Algorithm 187

5.3 The FIND-1 Data-Retrieval Module 191

5.4 The FIND-1 Data-Modification Module 212

5.5 The FIND-2 and FIND-3 Algorithms 242

CHAPTER 6: THE MAIN E-7 ALGORITHMS

6.1 Chapter Overview 259

6.2 The Concept of a Data-Image 261

6.3 The Basic E-4 Algorithms 269

6.4 The Basic E-5, E-6 and E-7 Algorithms 285

6.5 The Existential Quantifier, Universal Quantifier and Mean Value Algorithms 308

6.6 Multi-Variable Generalizations 312

6.7 The Maximum, Minimum and Median Value Algorithms 314

CHAPTER 7: EXTENSIONS

7.1 Chapter Overview 327

7.2 The Tabular Predicates and E-8 Algorithms 328

7.3 The Multi-Predicate Redundancy Removal Technique 346

7.4 Multi-Predicate Trade-off Techniques 354

7.5 Further Extensions 373

BIBLIOGRAPHY 382

1.1 Abstract

Let $e(x, y)$ denote an arbitrarily complex (nonatomic) predicate, and let $e(\bar{x}, y)$ denote the special form that this predicate will assume when the initial multi-component variable of x is replaced by the fixed tuple of $\bar{x}$. This thesis will study the question of how the set of y-records that satisfy the condition of $e(\bar{x}, y)$ can be efficiently retrieved. Such retrievals will be examined in the context of a "dynamic" environment where new records are being continually inserted, deleted, and updated. This thesis will show that every conceivable predicate of $e(x, y)$ can be assigned a database and a number of $D(e)$ such that

i) For any element of $\bar{x}$, the set of y-records that satisfy $e(\bar{x}, y)$ can be retrieved in $\Theta(\log^{D(e)} N)$ time (from the initial N-membered database).

ii) The relevant database will be sufficiently flexible so that the user can insert, delete, or modify any of its y-records in $\Theta(\log^{D(e)} N)$ time.

The above results will be extremely significant because the number of $D(e)$ will usually assume small values such as zero, one, or two.

The principle application of this thesis will be in the area of automatic programming. The purpose of that branch of computer science has been to discover how automatic algorithms can be developed which do much of the

programming that has traditionally been assigned to human beings.

Such automatic algorithms have been advocated by many computer scientists (Codd-70, Boehm-72, Date-77) because these procedures would dramatically reduce the cost of writing computer programs. The combined work of the cited authors have shown that

i) the cost of developing computer software may greatly exceed hardware costs in the 1980's (Boehm has estimated that computer programmer labor costs will constitute 90% of all the Air Force's 1985 computer related expenditures),

ii) and that the health of the computer industry requires lower software development costs (even if this is done in the context of a trade-off that modestly increases the hardware costs).

The importance of automatic database search algorithms was further confirmed in a recent panel discussion (Codd-75). The members of that panel concluded that such automatic search algorithms would be extremely useful if these algorithms could be made to be moderately efficient. This thesis will lay the foundations of the theory that should be used in the development of automatic predicate searching algorithms.

A formal description of the main theorems that will be proven in this thesis will be given in the next section.

1.2 Objectives of this Thesis

The purpose of this section will be to give a general description of the theorems that will be proven in this thesis. The discussion in this section will be divided into two parts. The first part will introduce the notation that will be used in the thesis, and the second part will use this terminology to offer a general description of the theorems that will be proven.

During this thesis, the symbols of x and y will be used to denote tuples. Also, the symbols of $x.a$ and $y.b$ will denote "attributes" which are associated with these tuples. These attributes will be defined to be either components of the cited tuples or arithmetic functions of them. For example, if $\bar{y}$ denotes the tuple of $(2, 4, 5, 8)$ then its attributes will include the cited four components as well as their arithmetic function such as, for example, "$(2 \times 4) + 6$". In general, the symbols of $x.a_j$ and $y.b_j$ will denote the j-th attributes which are associated with the respective variables of x and y.

During this thesis, it will be necessary to distinguish those occasions when x or y denote variables from the other occasions when they designate fixed elements. In general, a bar will be placed over these symbols when they denote fixed elements. Also, subscripted notation such as $\bar{y}_1 \bar{y}_2 \cdots \bar{y}_N$ will be used to designate a sequence of fixed elements. Obviously, this suggests that $\bar{y}_i.b_j$ will denote the j-th attribute of the i-th fixed y-element.

Throughout this thesis, it will be assumed that the variable of y will refer to a tuple that belongs to the set of R_y. The relational database terminology will often be used to describe this set. R_y will therefore be called a "relation", and it will be intuitively thought of being a table consisting of M columns (for the M specified attributes) and N rows (for the N distinct tuples that belong to R_y). Unlike most previous relational database articles, this thesis will allow two or more tuples in the R_y relation to contain the same information (because such a view of R_y is more general than the traditional relationalist perspective that has prohibited repetitions of identical tuples).

The symbol of $e(x, y)$ will be used in this thesis to denote an (nonatomic) predicate that describes the variables of x and y. In most of this thesis, it will be assumed that there are three types of atomic predicates which will be allowed to be contained in expression $e(x, y)$. These three possible classes of atomic predicates will be called the equality, order, and unary predicates. They are defined below:

Definition 1.2.A. An equality predicate will be defined to be an atomic predicate that consists of an equality comparison between two attributes of two distinct variables. (The prototype example of an equality predicate is "$x.a = y.b$.")

Definition 1.2.B. An order predicate will similarly be defined to be an order comparison between two attributes of two distinct variables. (The four prototype examples of order predicates are "x.a > y.b", "x.a ≥ y.b", "x.a < y.b" and "x.a ≤ y.b".)

Comment: Note that the preceeding two definitions required that order and equality predicates consist of comparisons between the attributes of "two distinct variables". This stipulation was placed in the preceeding definition because comparisons that only involve one variable will be defined in this thesis to be "unary" rather than "equality" or "order" predicates. The formal definition of "unary" predicates is given below:

Definition 1.2.C. A unary predicate will be defined to be any atomic predicate whose truth-value depends on only one variable. Furthermore, such a unary predicate will be defined to be "x-based" if it depends on the variable of x and y-based if it depends on the variable of y. (Some examples of unary predicates are "$y.b_1 < y.b_2$", "y.b > 10", "y.b = 10", "$y.b_1 = y.b_2$", and "x.a = 10".)

The main class of nonatomic predicates that will be studied in this thesis will be the "E-7 expressions". An E-7 expression will be defined to be any (nonatomic) predicate whose atomic components are unary, equality and order predicates. The principle goal of this thesis will be to develop a single algorithm that constructs and manipulates a database for all E-7 predicates.

In this thesis, it will ALWAYS be assumed that N denotes the number of y-elements that belong to the R_y relation. The main theorem of this thesis will state that the subset of R_y which satisfies the $e(\bar{x}, y)$ condition can be queried in $\Theta(\log^{D(e)} N)$ runtime. The exponent of D(e) in the previous equation will obviously be very important, and it is defined below:

Definition 1.2.D. The symbol of D(e) will be called the "degree of predicate e(x, y)", and it will be defined to be number of distinct y.b attributes which appear in expression e(x, y)'s order predicates.

Example 1.2.E. Let us calculate the D(e) values for the following E-7 expressions:

1) $\{x.a_1 < y.b_1 \text{ AND } x.a_2 > y.b_1 \text{ AND } x.a_3 \geq y.b_2 \text{ AND } x.a_4 \leq y.b_2\}$

2) $\{x.a_1 < y.b_1 \text{ OR } x.a_2 = y.b_2 \text{ OR } y.b_3 > y.b_4\}$

3) $\{x.a_1 = y.b_1 \text{ AND NOT } y.b_3 > 10\}$.

Equation 1 will have a degree equal to 2 because the four order predicates in that equation will process precisely two y-attributes (which are b_1 and b_2). Equation 2 will have a degree equal to one (because only its b_1 attribute is associated with an order predicate). Finally equation 3 will have a degree equal to zero (because neither of its atomic predicates are order predicates).

The concept of D(e) will be used during the analysis of the database

runtimes and memory requirements in this thesis. Throughout that discussion, the reader should bear in mind that a fully automized database procedure will necessarily contain four types of modules to complete its entire task. These four modules will be:

i) a compile module whose purpose will be to decide how the database for predicate $c(x, y)$ should be designed;

ii) a construction module whose purpose will be to physically construct this database;

iii) *a data-retrieval module* which will be activated by any argument of $\bar{x}$ and whose purpose will be to scan the cited database in order to gather the required information about the subset of R_y which satisfies $c(\bar{x}, y)$;

iv) a data-modification module which will require a tuple of $\bar{y}$ and an insertion (or deletion) command of c as arguments and whose purpose will be to take the record of $\bar{y}$ and either insert it (or delete it) from the relevant $c(x, y)$ database (in accordance with the user's initial insertion (or deletion) command of c).

All four of the preceeding modules will be discussed to some extent in this thesis. It should be stated most of the discussion in this thesis will center around the latter two modules. This is because the strategies which are used by the compiler and construction modules are obviously primarily related to

the goal of developing a "dynamic" database that supports efficient data-retrieval and data-modification operations.

Throughout this thesis, a two-component terminology will be used to describe the runtimes that are associated with the latter two modules. A given database task will thus be said to have an $(r_1 ; r_2)$ runtime if the first component describes the amount of runtime that is needed to perform retrieval operations and the second components describes the runtime that is needed to insert or delete a record in this database.

The second component of the preceeding runtime terminology will also have several implications beyond data-modification time. This is because the task of constructing a full N membered database can be performed by simply instructing the data-modification module to insert all N elements of R_y in a one-by-one manner. This fact means that the entire N membered database can be constructed in $N r_2$ runtime (if r_2 denotes the second component of the database's runtime). Thus, the general second components of database runtimes attain additional importance because they place bounds on the time that is required to construct these databases.

The r_2 component will also be important because it can be used to place a limit on the maximum amount of memory space that this database can occupy. This is because a database which requires $N r_2$ time to construct cannot possible occupy more than this amount of space. Thus, the preceeding

two paragraphs have shown that this thesis's proposed two-component runtime estimates will enable us to place bounds on data-construction time and memory space as well as measuring retrieval and data-modification time.

Most runtimes in this thesis will either be "worst-case" runtime estimates or "worst-case hash" runtime estimates. The former concept requires no definition since it has frequently been used in the computer literature. The latter concept has no close analog in the previous literature, and it therefore is defined below:

Definition 1.2.F. A predicate processing algorithm of A will operate in $(r_1 ; r_2)$ "worst-case hash" runtime if this runtime estimate is based only on the assumption that the relevant hash operation will distribute the data in the uniform manner that is normally associated with hashes. More specifically, if it is assumed that each hash search can be performed in one unit of runtime, then algorithm A will be said to have an $(r_1 ; r_2)$ worst-case hash runtime only if this runtime estimate holds rigidly for worst-case predicates, worst-case distributions of data, and worst-case changes in the data-base over time.

All the runtime estimates in this thesis will of course be order of magnitude approximations. The runtime coefficients in these approximations will have slightly different meanings than they had in previous computer literature. This is because very few previous researchers projects have attempted to estimate runtimes for arbitrarily complex predicates. It is

obvious that such measurements of runtime will be a function of both the size of the database and the nature of the specific predicate that is being processed. In this thesis, the runtime magnitude will be defined to be a function of the former term, and the runtime coefficient will be defined to be a function of the latter term. In other words, algorithm A will be said to have a runtime of $\Theta(r_1 ; r_2)$ magnitude if for every predicate of $c(x, y)$ there will exist a corresponding coefficient of k_c (whose value depends on predicate $c(x, y)$) and which satisfies the condition that the data-retrieval and modification tasks can be performed within the respective $k_c r_1$ and $k_c r_2$ runtimes.

Most of this thesis will center around the question about how to minimize the runtime magnitude since that topic is obviously more important than the optimization of the coefficient. The seven most important theorems that will be proven in this thesis are listed below:

Claim 1.2.G. Let $c(x, y)$ denote an B-7 expression, $\bar{x}$ denote a fixed element, F denote a function that maps R_y into an abelian group, and $S_c^F(\bar{x})$ denote the sum of the $F(y)$ values of the subset of R_y that satisfies $c(\bar{x}, y)$. It will be proven in this thesis that it is possible to design a procedure which will be called the *SUM* algorithm and whose purpose will be to take any element of $\bar{x}$ and calculate that element's $S_c^F(\bar{x})$ sum in $\Theta(\log^{D(c)} N ; \log^{D(c)} N)$

worst-case hash runtime. The detailed nature of the SUM algorithm can be understood if the meaning of the thesis's two component runtime terminology is recalled. The $\Theta(\log^{D(e)} N ; \log^{D(e)} N)$ runtime that is mentioned here will basically mean that for every predicate of $e(x, y)$ there will exist a corresponding database such that:

i) The retrieval module of this thesis's proposed SUM algorithm will be able to take any element of $\bar{x}$ and scan this database for the purposes of calculating $S_e^F(\bar{x})$ in $\Theta(\log^{D(e)} N)$ runtime (which of course represents the first component of the preceeding two part runtime).

ii) The data-modification module of the proposed algorithm can insert or delete any y-record in the preceeding database in $\Theta(\log^{D(e)} N)$ runtime (which of course corresponds to the second component of the preceeding runtime).

Claim 1.2.H. Let $e(x, y)$ once again denote an E-7 expression, $\bar{x}$ again denote a fixed element, and let $I_e(\bar{x})$ denote the number of y-records that belong to the subset of R_y which satisfies $e(\bar{x}, y)$. A straightforward modification of the SUM algorithm will be developed in this thesis that can calculate any $I_e(x)$ value in $\Theta(\log^{D(e)} N ; \log^{D(e)} N)$ worst-case hash time. The algorithm that performs this task will be henceforth called the COUNT algorithm.

Claim 1.2.I. Let $e(x, y)$, $\bar{x}$, and $I_e(\bar{x})$ have the same meaning as they did in the previous two claims. Let $Y_e(\bar{x})$ denote the subset of R_y that satisfies $e(\bar{x}, y)$. It will be proven in this thesis that the FULL LIST OF $I_e(\bar{x})$ DISTINCT ELEMENTS which belong to the $Y_e(\bar{x})$ set can be written into the user's workspace in $\Theta(\log^{D(e)} N + I_e(\bar{x}) ; \log^{D(e)} N)$ worst-case hashtime. The algorithm that performs this task will be henceforth called the FIND algorithm. It will also be proven in this thesis that the $I_e(\bar{x})$ component of the preceeding runtime will possess a runtime coefficient that is never greater than 2.

Comment. Note that the runtime of the FIND procedure differs from that of SUM and COUNT algorithms by containing an additional component of $I_e(\bar{x})$. The FIND algorithm will possess this additional runtime parameter because it will obviously require $I_e(\bar{x})$ units of additional time to transfer all $I_e(\bar{x})$ distinct members of the $Y_e(\bar{x})$ set into the user's workspace. The other four major database algorithms of this thesis are discussed by the following four claims:

Claim 1.2.J. *Let $e(x, y)$, $\bar{x}$, and $Y_e(\bar{x})$ have the same meaning as they had in the previous three claims. Let us additionally suppose that F maps the members of R_y into a field and that $M_e^F(\bar{x})$ denotes*

the product of the $F(y)$ values of the members of the $Y_e(\bar{x})$ subset (where multiplication is performed in this field's multiplicative semi-group). It will be proven in this thesis that it is possible to design a MULTIPLY algorithm that calculates such $M_e^F(\bar{x})$ products in $\Theta(\log^{D(e)} N ; \log^{D(e)} N)$ worst-case hashtime.

Claim 1.2.K. Let $e(x, y)$, $\bar{x}$, $Y_e(\bar{x})$, and $F(y)$ have the same meanings as in the previous claims. It will be proven in this thesis that it is possible to design a mean-value algorithm which calculates the mean $F(y)$ value of the members of the $Y_e(\bar{x})$ set in $\Theta(\log^{D(e)} N ; \log^{D(e)} N)$ worst-case hashtime.

Claim 1.2.L. Also, it will be proven in this thesis that it will be possible to design a "LOCATE" algorithm that can find the minimum, median, maximum or more general i-th smallest $F(y)$ value in the $Y_e(\bar{x})$ set in $\Theta(\log^{D(e)+2} N ; \log^{D(e)+1} N)$ worst-case hashtime.

Claim 1.2.M. This thesis will also show that it is possible to design existential quantifier and universal quantifier algorithms which will respectively determine whether the element of $\bar{x}$ satisfies the existential quantifier condition of "$\exists_y\, e(\bar{x}, y)$" and the universal quantifier conditon of "$\forall_y\, e(\bar{x}, y)$" in $\Theta(\log^{D(e)} N ; \log^{D(e)} N)$ worst-case hashtime.

There are several further remarks that should be made about the algorithms that satisfied the preceeding claims. Firstly, it should be pointed out that only the FIND algorithm had an $I_e(\bar{x})$ component governing its runtime. This fact means that the retrieval times of the SUM, COUNT, MULTIPLY, mean-value, median-value, minimum-value, maximum-value, LOCATE, existential quantifier and universal quantifier algorithm are completely unrelated to the number of y-records that satisfy the condition of $e(\bar{x}, y)$. In the example of the SUM procedure, it would thus take the same time to retrieve an $S_e^F(x)$ total when one y-record satisfies $e(\bar{x}, y)$ as when all N records satisfy this condition.

Secondly, it should be pointed out that all the runtimes in the preceeding claims were only order of magnitude estimates. Obviously a complete discussion of computer optimization would require an examination of both the runtime magnitudes and coefficients. The latter topic will not be discussed in great detail in this thesis partly because it would cause this thesis to reach an unwieldy length and also because the topic of optimizing the runtime coefficient is obviously far less important than the optimization of the runtime magnitude. As a general rule, the algorithms of this thesis will have a sufficiently small coefficient to insure that these algorithms will process typical commercial requests (that have half a dozen atomic predicates) with good efficiency.

It should also be stated that the algorithms of this thesis will frequently operate more efficiently than was indicated in the preceeding runtime estimates.

This is because the runtimes in this thesis are measured by the extremely conservative criteria of worst-case hashtime. In many instances, these algorithms will exceed their worst-case runtime estimates either because a special simplification will be applicable to a given predicate or because a given application will employ a more amenable then worst-case distribution of data.

The magnitude of the task that is being attempted in this thesis can be best appreciated if the examples of the predicates that are given in equations 4 through 6 are considered. If the reader pauses for a few minutes and examines these predicates in detail, he will see that there is no readily apparent algorithm that guarantees the worst-case hash runtimes that were mentioned in Claims 1.2.G through 1.2.M. If the reader does decide to examine the cited equations in detail, it should be kept in mind that an algorithm is not comparable to Claims 1.2.G through 1.2.M UNLESS BOTH ITS DATA-RETRIEVAL and DATA-MODIFICATION modules possess comparable WORST-CASE hash runtimes. The preceeding must be kept in mind because the relational data-base optimization problem becomes trivial and unimportant if one does not simultaneously consider data-retrieval and modification in the context of arbitrarily complex worst-case probability distributions. It should also be stated that the general difficulty of the predicate processing problem can be understood if equations 4 through 6 are considered in light of Claims 1.2.G's, 1.2.I's and 1.2.L's respective SUM, FIND and median-value algorithms.

4) $\{x.a_1 < y.b_1 < x.a_2 \& x.a_3 < y.b_2 < x.a_4\}$

5) $\{x.a_1 < y.b_1 \text{ OR } \{x.a_2 < y.b_2 < x.a_3 \& x.a_4 \neq y.b_3\}$

6) $\{x.a_1 < y.b_1 \& x.a_2 \neq y.b_2 \& x.a_3 \neq y.b_3 \& x.a_4 \neq y.b_4\}$.

The challenge that must be overcome in developing a predicate searching algorithm was discussed in Knuth-73. Knuth explicitly mentioned the above equation 4 as an example of a previously unsolved predicate. In his textbook, Knuth stated that it would be extremely important if someone developed a universal search algorithm that would solve general predicates (see pages 550 through 555). Obviously, the algorithms of this thesis will constitute most of the solution of Knuth's problem. Indeed, the algorithm of this thesis will transend Knuth's initial suggestion because Knuth's proposal was made mostly in the context of developing an algorithm that satisfies a condition similar to the FIND procedure that was mentioned in Claim 1.2.I. The other six claims of this section basically go beyond Knuth's initial research suggestion.

It should also be stated that the preceeding Claims 1.2.G through 1.2.M only offer a partial description of the goals of this thesis. Thus, many additional theorems will be proven that go beyond these claims. For example, all of Chapter 7 will be devoted to further generalizations of this subject matter. The topics of Chapter 7 will include:

i) the "E-8 expressions" which are a generalization of the E-7 class that will include a fourth type of atomic predicate (which is called the tabular predicate);

ii) the discussion of how to develop a compound database that simultaneously serves several predicates;

iii) and the application of the general algorithms of this thesis to Codd's relational calculus.

There will also be still other further important topics and theorems in this thesis. The most important among these topics will be Chapter 3's introduction of the concept of a "super-B-tree". These super-B-trees will be crucial to virtually all the algorithms of this thesis. The first three pages of Chapter 3 will offer a brief summary of the goals and the nature of the super-B-tree algorithm.

1.3 An Intuitive Explanation of the Main Algorithms

There will be three basic themes that will intuitively motivate most of this thesis. A description of these three themes will be given here.

The first theme will be that the COUNT algorithm has many useful applications. This algorithm was defined in Claim 1.2.11 of the previous section. That claim indicated that the COUNT algorithm would efficiently calculate a number, denoted as $I_e(\bar{x})$, which indicated how many y-records satisfied $e(\bar{x}, y)$. The COUNT algorithm will be important because it will provide the database management system with information describing the statistical distribution of the data. This statistical information will enable the database management system to determine which of several eligible search methods will most efficiently retrieve the y-records that satisfy $e(\bar{x}, y)$. The technique of using the COUNT algorithm to determine a search method will produce improvements in the runtimes of the FIND, existential quantifier, universal quantifier, minimum value, mean-value, median value, maximum value and LOCATE algorithms. In each of these cases, the COUNT algorithm will provide statistical information which indicates how these procedures can be rapidly executed.

The second theme of this thesis will be that the COUNT algorithm is a special case of the more general SUM algorithm. The reason a large part of this thesis will be devoted to the SUM algorithm will be that this procedure

will immediately enable one to calculate COUNTs.

The third theme of this thesis will center around the many uses of "super-B-trees." These trees will be employed by every major algorithm in this thesis. Their purpose will be to form a generalization of traditional sorting structures that help process complicated combinations of order predicates. An introductory description of super-B-trees will be given in section 3.1. Some readers may wish to immediately advance to that section so that they can examine the formal definition of these trees.

1.4 Organization of the Thesis

The purpose of this section will be to provide a general description of how the subject matter in this thesis has been organized. It should be first stated that the main algorithms of this thesis have been carefully divided into approximately two dozen subroutines. This division of the subject matter has been done in the interests of a compartmentalized presentation that will make it easier to prove the theorems.

During this thesis, there will be eight classes of predicates that will be studied. These eight classes of predicates will be called the E-1 through E-8 expressions. Each class of E-(k + 1) predicates will be a generalization of its E-k predecessors. Also, the algorithms for E-(k + 1) predicates will be generalizations of their E-k predecessors. Typically, these higher level E-(k + 1) algorithms will make subroutine-calls to the lower level E-k procedures. This step-by-step discussion of gradually more complicated predicates will make it easier to prove the main theorems of this thesis.

Let k denote any integer whose value is between 1 and 8. For each such k, this thesis will define three corresponding "SUM-k", "COUNT-k" and "FIND-k" algorithms. These algorithms will of course be designed to perform the respective SUM, FIND, and COUNT calculations for their respective classes of E-k expressions. The most important objective of this thesis will be to develop the SUM-7, COUNT-7 and

FIND-7 algorithms that process E-7 expressions in the manner suggested by Claims 1.2.G through 1.2.I.

Many of the paragraphs in this thesis will be given the standard mathematical headings of "Theorem", "Proof", "Definition", etc. Also, some of the paragraphs will be given special labels, such as "Remark", "Observation" and "Comment". There will be a very important distinction in this thesis between these latter three types of headings. It will be very useful for the reader to be aware of this distinction, and it is explained below:

i) A paragraph will be given the heading of "Remark" if it describes an extension of the main algorithms of this thesis whose purpose will be to go beyond the central claims that were mentioned in the previous section. The principle topics of these Remarks will be the techniques that optimize the runtime coefficient. The Remarks should be regarded as optional reading. They are not related to the main theorems, and they contain primarily technical information about how the main concepts of this thesis can be further extended. The discussion in these "Remarks" has been kept short and free of proofs for the sake of a briefer presentation.

ii) A paragraph will be given the heading of "Observation" if that paragraph contains a statement which is so trivial that it requires no proof. All "Observations" in this thesis are given identification

numbers, and several theorems in this thesis will cite these observations during their proofs.

iii) A paragraph will be given the label of "Comment" if that paragraph contains the usual clarifying information that the mathematical literature has normally associated with this term.

Once again, it should be emphasized that the reader will benefit if he remembers the distinction between "Remarks", "Observations" and "Comments". Obviously, the most important aspect of this distinction is that the "Remarks" should be regarded as optional reading that contains highly technical information.

There will be six main chapters in this thesis. Chapter 2 will provide a summary of the previous attempts to develop algorithms similar to those that are proposed here. It should be stated that Chapter 2's review of the previous database literature is not directly relevant to this thesis because the cited articles have not relied upon highly mathematical techniques. The reader should bear this in mind as he examines Chapter 2.

A further discussion of the previous computer literature will take place in the beginning of Chapter 3. The early parts of Chapter 3 will discuss the previous B-tree literature. The remainder of Chapter 3 will propose a new tree-manipulating algorithm that is called the "super-B-tree" procedure. The concept of a super-B-tree will be extremely important.

The main algorithms of this thesis will be presented in Chapters 4, 5, and 6. Chapter 4 will present the simplest versions of the SUM and COUNT algorithms. Chapter 5 will discuss the similar low-level version of the FIND algorithm, and it will also explain the relation between the FIND and COUNT procedures. Chapter 6 will generalize the preceeding results for E-7 expressions. That chapter will also contain the proofs of the previous Claims 1.2.G through 1.2.M.

The last major part of this thesis will be Chapter 7. That chapter will discuss various extensions of the main E-7 algorithms. The general nature of these extensions was also briefly mentioned at the end of section 1.2.

It should be stated that the material in this thesis has been very carefully organized. All the chapters of this thesis will begin with an introductory section that describes the theorems which will be proven in that chapter. The reader can examine the six main chapters of this thesis in almost any order he desires after he has examined the introductory sections of these six chapters. Among the many possible orders for reading this thesis are:

i) Chronological order (which has arranged the theorems so that each successor theorem is a consequence of its predecessors).

ii) Order where the most important theorems are examined first. In this case, the thesis should be read in two stages. In the first stage, all of Chapter 3, sections 4.1, 5.1 and Chapter 6 should be

examined. In the second stage the remainder of this thesis should be read.

iii) Order where the simplest material is read first. Again, in this case, the thesis should be read in two stages. In the first stage, sections 3.1 through 3.3, all of Chapter 4, sections 5.1, 5.2, 5.5 and all of Chapter 6 should be read. In the second stage, the remainder of this thesis should be examined.

iv) Order for those readers who wish only to skim this thesis. In this case, the reader should only thoroughly examine sections 3.1 through 3.3, 4.1, 5.1, and Chapter 6.

Two final remarks should be made about the above four reading orders. First, it should be stated that the reason Chapter 2 was not mentioned in any of the reading orders is that this chapter's description of the previous database literature is basically unrelated to the highly mathematical techniques of this thesis. Consequently, Chapter 2 can be examined either before or after the remainder of this thesis.

Secondly, it should be pointed out that all four of the preceeding reading orders begin with the same sections 3.1 through 3.3. Consequently, the reader can postpone his decision about how to read this thesis until after these first three sections have been examined.

2.1 General Description of Previous Research

Interest in the topic of automatic database algorithm similar to those proposed in this thesis began around the year of 1970. At that time, Codd wrote a series of articles where he advocated such systems (Codd-70, Codd-71a, Codd-71b). The theme of Codd's work was that the nature of computers was radically changing as a result of new less expensive computer hardware technologies. Codd predicted that the falling cost of computer hardware would make it increasingly possible for an equal improvement to be made in cost of computer programming. Codd advocated a new computer language that would enable the user to access his database through goal-oriented primitives rather than the current procedural primitives. Codd felt that such procedural primitives were harmful because they caused the computer programmer to waste his time by planning the detailed nature of the required search tasks. Codd argued that searching could be automated by a single algorithm and that this algorithm would conserve the precious time of the human programmer. Codd did not indicate how such a database management system should be designed. Instead, he proposed two prototype languages (called the relational calculus and algebra) and urged the computer science community to consider how these languages should be efficiently implemented.

Codd's work has generated a great deal of interest in the computer science community. The magnitude of this interest can be seen upon examination

of Chamberlain-76. That article has a bibliography which lists 171 references on the subject of relational databases. This bibliography only covers the 1970-1975 period. Obviously, a large number of additional articles have been written in the last two years.

An examination of the 171 references that are given in Chamberlain's bibliography as well as the other more recent computer literature reveals that only a small percentage of these articles have addressed themselves to the question of how a relational database management can be made to generate efficient codes. Instead, most of these articles have discussed improved relational computer languages, security under relational database systems, the added expressibility of normalized relations, etc. The small amount of research into efficiency can be easily seen if the bibliography in Chamberlain-76 is examined in detail. That bibliography has arranged its various references under approximately ten different headings. Only 25% of the references in this bibliography have been placed in categories related to runtime performance. A reading of these articles as well as other more recent references reveals that no one has previously discovered how to make a relational database management system operate with considerable efficiency. A further indication about the previous research into the efficiency of relational database management systems has been given in BE-77. In the third paragraph of that article, Blasagan and Eswaran state that "little" has been published about the subject of efficiency. It should be stated that BE-77 was published in the last quarter of 1977. It thus

offers an up-to-date account of the previous research into the efficiency of automatic relational database management systems.

One final prefatory remark should be made before a detailed description is given to the previous relational optimization literature. It should be stated that the goals of the previous projects differed markedly from thos in this thesis. The principle differences are that:

1) Most of the previous articles on relational databases ignored theorem proving and instead emphasized the development of an experimental prototype. In contrast, the exclusive emphasis here will be on theorem proving.

2) Because of their experimental orientation, most of the previous projects confined themselves to simple predicates and were further forced to consider the distinction between main and disc memory. The emphasis here will be in the exact reverse. The topic of auxiliary memory will be postponed to a later time. Instead, the main topic will be the study of how complicated predicates can be assigned efficient runtime magnitudes.

3) Also, runtime is measured differently in this thesis than it was in the previous relational literature. Most previous measurements of runtime only held for certain predicates and for uniform probability distributions of data. In contrast, the "worst-case hash" runtime in this thesis will hold under all conceivable circumstances.

The reader should keep in mind the preceeding three points as he examines the detailed summary of the previous relational database literature that is given in the next section. It should also be kept in mind that the sole purpose of the next section is to explain the challenges that confronted previous relational projects. Frankly, the previous relational database literature will not be used during the detailed theorem proving that which takes place in Chapters 3 through 7. Instead, these theorems will be an outgrowth of the previous B-tree literature (which is summarized in section 3.2).

2.2 A More Detailed Description of Previous Research

There have been three main languages that have been proposed by the advocates of relational databases. These languages have been called the relational calculus, the relational algebra, and the relational mapping languages. A brief summary of the optimization techniques that have been applied to these languages will be given in this section. That discussion will be divided into three parts which separately discuss how each of these three different languages have been previously optimized.

This discussion will begin with the relational calculus. This language is somewhat similar to the predicate calculus. The relational calculus language possesses the same universal and existential quantities that have been associated with the predicate calculus. The only difference between these languages is that the relational calculus quantifiers range over finite user-created sets (called "relations") rather than the infinite membered sets of the predicate calculus.

There have been very few articles written about how to implement the relational calculus because it is obviously very difficult to develop an algorithm that processes all conceivable relational calculus expressions in efficient run-time. Some articles have been written by Rothnie.

In his articles, Rothnie assumed that the variables of x and y spanned two separate relations (that will be denoted as R_x and R_y in this thesis). The goal of Rothnie-74 and Rothnie-75 was to find the subset of R_x

that satisfied equation 1 without engaging in a costly exhaustive search through the entire cross-product set that is generated by R_x and R_y.

1) $$\{\exists y\ e(x, y)\}$$.

Rothnie showed that a costly search through the large XY-cross-product set was unnecessary, and that the search space could be restricted with the following 3 optimizing modules:

1) If one y-element is found which satisfies $e(\bar{x}_i, y)$, then Rothnie's first module will automatically stop searching for other y-elements that could conceivably also satisfy this predicate (since at that point, it has been determined that $\bar{x}_i$ meets the condition indicated by equation 1).

2) If a y-element is found which satisfies $e(\bar{x}_i, y)$ <u>and if</u> a theorem prover has verified the statement:

2) $$\{\forall y[e(\bar{x}_i, y) \rightarrow e(\bar{x}_k, y)]\}$$

then Rothnie's second module will automatically conclude that $\bar{x}_k$ also satisfies the preceeding equation 1.

3) If Rothnie's algorithm discovers that NO y-elements can possibly satisfy $e(\bar{x}_i, y)$ <u>and if</u> a theorem prover has verified the statement:

3) $$\{[\forall y \text{ NOT } e(\bar{x}_i, y)] \rightarrow [\forall y \text{ NOT } e(\bar{x}_k, y)]\}$$

then without further examining the element $\bar{x}_k$, Rothnie's third module will automatically conclude that $\bar{x}_k$ cannot possibly satisfy equation 1.

Rothnie has indicated that his optimizer's second and third modules are optional. He tested his optimizer via simulation, and the results showed that the second and third modules may improve or hamper efficiency, depending on the nature of the predicate e. Rothnie's articles contain no theorems or proofs, and he discusses only simulation results. Rothnie's optimizations are clearly useful, but in many cases the algorithm is still inefficient. One can for instance construct worst-case examples where if everything goes wrong, Rothnie's algorithm will require N^2 time to process such expressions as simple as:

4) $$\{\exists y : x > y\} \quad .$$

There have been very few additional articles written about the relational calculus since Rothnie's original articles. The difficulty with relational calculus optimization can be seen if Chiba-75 is examined. In that article, Chiba

i) evaluates existential quantifiers with a procedure similar to Rothnie's module 1

ii) evaluates universal quantifiers with the obvious analagous

procedure that exhaustively scans the R_y relation until a y-element is found that does not satisfy the condition of $e(\bar{x}_i, y)$

iii) evaluates nested sequences of quantifiers (similar to "$\forall y \exists z \forall w\, e(x, y, z, w)$") with the generalization of these procedures that uses a nested sequence of DO-COOPS to process the nested sequences of quantifiers.

The disadvantage of the above algorithm is of course that it requires $\Theta(N^M)$ runtime when it processes M quantifiers (in the worst-case). The universal and existential quantifier procedures that were previously mentioned in Claim 1.2.M of this thesis will obviously be a considerable improvement over the previous relational calculus algorithms. It should also be stated that sometime in the coming year, I plan to write a paper that will show how most M-quantifier relational calculus expressions can be processed in $\Theta(N \log N)$ runtime. The specific theorems that are planned to appear in this coming paper are listed in section 7.5 of this thesis.

The second topic of the relational optimization literature has been the relational algebra language. This language has been designed to contain several important set-manipulating commands. The primitives in this language include commands to take the union, intersection, cross-product, and difference between two sets as well as some other more obscure commands such as "selection", "join", "projection" and "set-division." The relational algebra

selection command has been defined to be a primitive that requests the subset of R_y that satisfies the unary predicate of $U(y)$ (when the user makes a selection command with these two arguments). The relational algebra join command is a generalization of the selection command that has been defined to be a primitive that requests the subset of the cross-product of R_x and R_y that satisfies $e(x, y)$. The relational algebra projection and set-divisions commands will not be defined here because they are unrelated to this thesis.

There have been two basic optimization topics that have been discussed in the relational algebra literature. The first topic has been techniques that transform one relational algebra expression into an equivalent expression that executes more efficiently. This topic has been explored in Palermo-72, Percher-75 and SC-75. The cited articles are not relevant to the predicate processing question that will be examined in this thesis. They will therefore not be discussed here.

The second topic of the relational algebra literature has been the question of how to efficiently execute join operations. The goal of this study has been to develop an algorithm that finds the set of xy-ordered pairs which satisfy $e(x, y)$ without making an exhaustive search through the N^2 distinct members of the xy-cross-product set. This topic was briefly discussed in Palermo-72, and it was subsequently examined in further detail in Gotlieb-75 and BE-77. The latter article discussed this topic in the greatest generality. The details Blasagan and Eswaran's work is therefore described below.

Let $U_1(x)$ and $U_2(y)$ denote respectively an x- and y-based unary predicate. Blasagan and Eswaran have shown how joins can be efficiently performed for predicates similar to those shown in equation 4

4) $$\{U_1(x) \text{ AND } x.a = y.b \text{ AND } U_2(y)\} \quad .$$

The discussion in BE-77 presents four different types of algorithms that can be applied to the preceeding predicate (under the assumptions of different probability distributions and different data-structures that are used to represent the R_x and R_y relations). The optimizer in BE-77 attempts to find the best combined utilization of disc and main memory. The main limit in the discussion in BE-77 is that the article has restricted itself to considering only very simple predicates similar to equation 4.

The last class of languages that has been studied by the relationalists is the relational mapping languages. Two of the most important relational mapping languages are QUEL and SEQUEL. The former language has been developed by the Berkeley Electrical Engineering Department and the latter language by the IBM San Jose Research Center. Articles discussing optimization techniques in these languages include HSW-75, WY-76, AC-75 and San Jose-76.

The SEQUEL and QUEL research papers have generally discussed different optimization topics. The distinction is that the SEQUEL papers have discussed optimization in a one and two variable setting whereas the QUEL papers have considered optimizations that involve requests with a larger number of variables. Only the SEQUEL optimizations will be discussed here because

SEQUEL's emphasis on one or two variables more closely corresponds to the similar one and two variable emphasis in this thesis.

The first paper on the subject of SEQUEL optimization was AC-75. In that paper, Ashtran and Chamberlain recommended using inverted files to find the set of y-records that satisfy a general predicate that will be denoted here as $e(\bar{x}, y)$. An example of such an application can be seen if the predicate in equation 5 is considered

5) $$e(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 \neq y.b_2\} \quad .$$

An inverted file on the y-attribute of b_1 would enable Ashtran and Chamberlain to easily locate all the y-records that satisfy $y.b_1 = \bar{x}.a_1$. Subsequently, an exhaustive scan through this set of y-records could be used to find its special subset that satisfies the additional condition of "$x.a_2 \neq y.b_2$". Obviously, this method will lead to a procedure that locates the y-records that satisfy equation 5 in usually efficient runtime.

The weakness of the above procedure can be seen if one considers how this algorithm will respond if the relevant database is governed by a probability distribution which is not uniform. For example, consider the case where

i) 89% of the y-records in the database have $y.b_1 = \bar{x}.a_1$ and $\bar{y}.b_2 = x.a_2$.

ii) 1% of the y-records in the database have $y.b_1 = \bar{x}.a_1$ and $y.b_2 \neq \bar{x}.a_2$.

In this case, the preceeding inverted file algorithm will search 90% of the database to locate 1% of its records.

The problem that has been outlined in the previous paragraph may at first appear to be unimportant since such circumstances may appear to be improbable. Unfortunately, such appearances are deceptive because real world database applications are usually governed by probability distribution which are much more complicated than uniform probability distributions. The basic weakness of inverted files is thus that one simply cannot assume a uniform probability distribution for an automatic database system that is designed to serve all classes of real world users.

It should be stated that AC-75 was intended to only be an exploratory article. The article omits mentioning many important subjects (such as for example sorting), and it obviously is not intended to offer a definitive work on relational optimization. The importance of AC-75 is that it was the first work that the IBM San Jose Research Center published on SEQUEL optimization.

The subsequent computer literature indicates that the IBM San Jose Research Center went to great lengths to try to develop a successful computer prototype that would support the SEQUEL language. The extent of this effort can be seen if San Jose-76 is examined in detail. That article contains the

names of fourteen coauthors who participated in the planning of this project.

The discussion in San Jose-76, indicates that San Jose's System R project is of course employing a much more powerful optimizer than was illustrated in AC-75. This optimizer allows the user to supply the database management system with useful optimization hints, such as recommended sorted lists and links. (The latter term refers to special automatically maintained pointers that indicate which ordered pairs of records satisfy some user specified condition.) Also, San Jose-76 indicates that System R project has attempted to distinguish between the costs of main and disc memory.

Unfortunately, it is very difficult to make further comments about the specific nature of the optimizations that are employed by System R because IBM has published very little information about this subject. For example, only four of the forty pages in San Jose-76 discuss the optimizations which are used by System R. That discussion centers around two examples of simple predicates. No general rules about how to efficiently process complicated predicates can be inferred from these examples.

There has been very little additional information published about the System R optimizer. Some articles on this subject that have appeared during the last six months were published by Blasagen, Casey and Eswaren (BE-77 and BCE-77). The first of these articles was mentioned earlier in this section because it used the terminology of the relational algebra language.

It was thus indicated that BE-77 showed how to efficiently locate the xy-ordered pairs that satisfy predicates of the form of:

6) $$\{U_1(x) \text{ AND } x.a = y.b \text{ AND } U_2(y)\} \quad .$$

Blasagen and Eswaran stated that their article was motivated by the research that was previously done on the System R optimizer.

The second recent article by the San Jose Research Center (BCE-77) discussed applications of Lum's multi-key sorted lists to relational databases (LUM-70). It is well known that such sorted lists can be used to efficiently find in $\Theta(\log N)$ time the set the y-records that satisfy conditions such as

7) $$x.a_1 = y.b_1 \,\&\, x.a_2 = y.b_2 \,\&\, \cdots\, x.a_j = y.b_j \,\&\, x.a_L \leq y.b_K \leq x.a_M \quad .$$

The main concern of Blasagen, Casey and Eswaran was to develop techniques that would minimize the coefficient that is associated with $\Theta(\log N)$ searches into Lum's multi-key sorted lists.

It should be stated that there is a very distinct difference in emphasis between the previous relational database research and that in this thesis. The distinction is that the previous relational research has concentrated on minimizing the runtime coefficient whereas the interest here will be in optimizing the runtime magnitude.

As the reader examines this thesis, he should bear in mind that the

runtime magnitude problem has been previously solved only for simple predicates that are either roughly similar to equations 6 and 7 or are disjunctions of such predicates. If $e_1 e_2 \cdots e_L$ denotes a sequence of predicates similar to equations 6 and 7 then equation 8 is an example of a disjunction predicate that can also be efficiently solved

8) $e_1(x, y)$ OR $e_2(x, y) \cdots$ OR $e_L(x, y)$.

Obviously, the set of y-records that satisfy equation 8 can be found with reasonable runtime magnitude if one simply takes the union of the set of records that satisfies the $e_i(x, y)$ predicates (since L will usually be a small number).

The main omission of the previous database literature has been the processing of more complicated predicates in the setting of possibly unusual probability distributions. These complicated predicates differ from the simple predicates by possessing "NOT" symbols (as in "x. a ≠ y. b") and by containing phases that consist of complicated conjunctions of order predicates (as for example equation 9)

9) $\{x.a_1 < y.b_1 \text{ AND } x.a_2 < y.b_2 \text{ AND } x.a_3 < y.b_3\}$.

Once again, it should be stated that the difficulty in querying arbitrarily complicated predicates in the context of all conceivable probability distributions was recognized in Knuth-73 (pages 550 through 555). Knuth stated that this would

be a very important problem to solve. It should also be repeated that this thesis will process complicated predicates in a manner that optimizes worst-case retrieval and worst-case data-modification time. Both must be considered for the problem to become nontrivial.

There frankly has been very little discussion of complex predicates in the previous relational database literature. This is because relational optimization theory is a very new subject. For example, all the articles mentioned in this literature survey were published in the last five years. The nature of the previous relational database optimization research was summarized in BE-77. The authors state that despite the fact that a large number of articles were published about the virtues of relational database languages and despite the fact that four major efforts were made to implement such a language on a computer (they cite Berkeley-75, Toronto-75, San Jose-76, England-76), it is nevertheless true that "little has appeared on the (runtime) performance of such systems" in the computer literature. This statement was published in the last quarter of 1977. It thus offers an up-to-date account of the previous optimization research.

The purpose of this thesis will be to develop a very large part of the mathematical foundation that will be needed by a future relational database management system. The theory that will be presented in this thesis will be an extension of B-tree theory. The previous B-tree research that pertains to this thesis will be discussed at the beginning of the next chapter.

3.1 Chapter Overview

The term of "B-tree" has been used in the computer literature to describe a tree representation of a sorted list that is sufficiently flexible to guarantee that any record can be inserted deleted, or retrieved from the B-tree in $\Theta(\log N)$ time. This chapter will introduce an important new combinations technique called the "super-B-tree." This tree will possess all the characteristics that are normally associated with B-trees plus one additional feature as well. This additonal property can be most easily explained if the interior nodes of traditional B-trees are examined in detail. Note that these interior nodes had a very simple structure and that they merely contained that information which was necessary to maintain an (efficient) chain of pointers connecting the tree-root to each leaf of the B-tree. The super-B-trees will have one additional field of information in their interior nodes. That additional field will be a pointer to a "subtree description structure" (SDS). The user will be allowed to store any information he desires in these SDS fields. The advantage of the proposed super-B-tree algorithm will be that it will manipulate these SDS structures in a highly efficient manner. A more detailed description of the nature of super-B-trees and their SDS fields will be given in the next several paragraphs.

The SDS field of node v will generally be assumed to contain any

type of information which the user desires to store about those leaves which are a descendant of v. For example, in some applications the SDS field of v may contain a hashed list that describes the descendants of v. In other applications, the SDS field may be a B-tree itself that differs from the super-B-tree insofar as the second B-tree has arranged v's descendants in a different order by a different Key value. In still other applications, the SDS of node v may contain the sum of the leaves that descend from v. In general, the SDS fields of the various nodes of our super-B-trees will be allowed to contain any information which the user desires to store about the descendants of these nodes.

The advantage of the proposed super-B-tree algorithm will be that it will manipulate the SDS fields in a very efficient runtime. This runtime can be best described if w denotes the amount of runtime that is required to update the SDS field whenever a node gains or loses a descendant leaf. The main theorem in this chapter will state that any leaf can be inserted or deleted from a super-B-tree in $\Theta(w \log N)$ WORST-CASE time.

The proof of this super-B-tree theorem and the procedure employed by it are very different from their traditional B-tree counterparts. The reason for this distinction can be understood if traditional B-trees are examined in further detail.

Traditional B-trees will not operate efficiently if an SDS field is

added to them because this field will cause the WORST-CASE cost of B-tree insertion and deletion operations to rise above $\Theta(w \log N)$ time. The reason for these difficulties can be understood if the case is considered where the modification of one or two pointers causes $N/4$ leaves to gain a new ancestor. In that case, expensive $wN/4$ time will be needed to modify the corresponding SDS field of traditional B-trees. These B-trees are thus unable to efficiently employ SDS structures because they too frequently engage in such pointer changing operations. Only the super-B-tree algorithm will be shown to insure $\Theta(w \log N)$ runtime even in WORST-CASE circumstances.

The discussion in this chapter will be divided into six sections. Section 3.2 will review the B-tree literature and compare the traditional B-trees from the perspectives of four different measuring criteria. By the end of Section 3.2, the reader will see that a marked improvement in runtime can be attained if a slight modification is made in Nievergelt and Reingold's bounded balance algorithm.

The super-B-tree algorithm will be introduced in part 3.3 of this chapter. That discussion will explain the intuitive idea behind the super-B-tree algorithm. By the end of section 3.3, the reader will see the connection between Nievergelt's and Reingold's previous work and my proposed super-B-tree algorithm.

The formal description of the super-B-tree algorithm will be presented

in sections 3.4, 3.5, and 3.6 of this chapter. Section 3.4 will formally define the super-B-tree algorithm, and sections 3.5 and 3.6 will respectively prove the correctness and $\Theta(w \log N)$ runtime of this procedure.

One final prefatory remark should be made before the detailed discussion begins. The reader should be reminded that the material in this thesis has been carefully arranged so that the five main chapters of this thesis can be read in any order after the reader has examined the first section of each chapter. At this point in time, the reader therefore has the option of either reading this chapter's detailed description of super-B-trees or proceeding to another chapter.

3.2 An Examination of B-Tree Theory

This section will study the implications which an SDS field has on traditional B-tree algorithms and the implications which it has on certain modified B-tree algorithms (that are proposed in this section). This discussion will include an examination of "AVL", "2-3", "multiway", "bounded balance" and other more general forms of B-trees. The runtimes of these procedures will be measured from the points of view of four different criteria. The challenge which an SDS field presents to a traditional B tree algorithm will be vividly illustrated in this section. By the end of the section, the reader will also begin to see the connection between super-B-trees and bounded balance trees.

A description of traditional B-tree algorithms can be found in the computer literature in references such as AVL-62, Knuth-73, NR-74, and AHU-74. A more summarized descripiton of B-trees will be given in the next several paragraphs.

In this section, the reader should bear in mind that all four types of B-trees will support insertion, deletion and retrieval operations that can be executed in the same $\Theta(\log N)$ worst-case runtime. The reader should also bear in mind that the only major distinction between these four classes of B-trees is that they will use different internal data-structures for representing their sorted lists.

The first article about B-trees was written by Adel'son-Velskiy and Landis in 1962 (AVL-62). The trees mentioned in that article have subsequently been called the AVL trees. The AVL trees are defined to be binary trees* where every pair of brothers are required to have heights which differ by no more than an integer of 1. The AVL trees are important because Adel'son-Velskiy and Landis showed that they support $\Theta(\log N)$ insertion, deletion, and search operations. A very clear discussion of the properties of these trees can also be found in Knuth-73.

A second type of B-tree algorithm was proposed by Ullman. These trees are called "2-3 trees," and they are defined by the two conditions that:

i) every interior node has either 2 or 3 sons;

ii) all leaves in the 2-3 tree have the same depth.

Ullman showed that the 2-3 trees will support the same $\Theta(\log N)$ insertion, deletion, and search times that are also associated with all other types of B-trees. A very clear description of the properties of 2-3 trees can be found in AHU-73.

A third type of B-tree is the "multi-way" trees. The multi-way trees are a generalization of the 2-3 trees. The multi-way trees will require that the user supply a parameter of m, and the multi-way algorithm will subsequently produce a tree whose interior nodes have between m and 2m-1 sons.

* A binary tree is defined to be a tree such that every interior node has precisely two sons.

A brief description of multi-way trees is given in Knuth-73. The proofs about the correctness and runtime-magnitude of multi-way trees are identical to those proofs which describe 2-3 trees. These two types of trees differ mainly in their runtime coefficients. The multi-way trees will have a preferable coefficient in those applications where computer files are stored on disc, and the 2-3 trees will be prefereable when the file is stored in main memory.

A fourth type of B-tree has been proposed by Nievergelt and Reingold (NR-74). Their trees are called "bounded balanced" trees. This chapter's *super-B-tree algorithm will be an expanded version of the Nievergelt-Reingold* algorithm. Consequently, a detailed description of bounded balance trees will be given in the next several paragraphs. This discussion will begin with several definitions.

Definition 3.2.A. Let v denote some node of a binary tree. The symbol of N_v will denote the number of leafs which are a descendant of node v. (Also, N_v will be defined to be given the default value of 1 when v is a leaf.) Throughout this chapter, N_v will be called the "leaf-weight" of node v.

Definition 3.2.B. Let v denote an interior node of a binary tree, v_L denote its left son, and let N_v and N_{v_L} denote the leaf weights of these two nodes. The symbol of $p(v)$ will henceforth denote that quotient which is produced when N_{v_L} is divided by N_v. This symbol will be called the

"balance" of node v in this thesis. (Nievergelt and Reingold use a slightly different terminology to describe the same concept. They use the term of "root balance" rather than the "balance" to describe this notion. A different terminology is used in this chapter because it lends itself more easily to the theorems which will be proven.)

There are several different classes of bounded balanced trees which are discussed in NR-74. The authors define a separate class of $BB(\alpha)$ trees for every value that the real number of α can assume when it varies between zero and 1/2. The definition of such a class of $BB(\alpha)$ trees is given below:

Definition 3.2.C. The symbol of $BB(\alpha)$ will denote the subset of binary trees whose every interior node of v will necessarily satisfy the inequality of

1) $\alpha \leq p(v) \leq 1 - \alpha$.

Comment: A less formal, more intuitive definition can be given to the class of $BB(\alpha)$ trees. Let v denote any node in a $BB(\alpha)$ tree, v_L denote the left son of v, v_R denote the right son of v, and let N_v, N_{v_L} and N_{v_R} denote the respective leaf-weights of these three nodes. Intuitively, the $BB(\alpha)$ trees will represent that subclass of binary trees whose N_{v_L}/N_v and N_{v_R}/N_v ratios are greater than α for every node, v, of these trees.

Let α denote any real number that satisfies the inequality of:

2) $$0 < \alpha \leq 1 - \sqrt{2}/2 \quad .$$

Nievergelt and Reingold showed that for each such α there will exist a corresponding B-tree algorithm that manipulates the set of $BB(\alpha)$ trees in $\Theta(\log N)$ runtime. Nievergelt and Reingold did not give any special name to their B-tree algorithms. In this thesis, the term of "$Alg(\alpha)$" will refer to the B-tree procedure that Nievergelt and Reingold would recommend for manipulating $BB(\alpha)$ trees. The specific results which were obtained by Nievergelt and Reingold were that

I) $Alg(\alpha)$ will need only $\Theta(\log N)$ time to search any $BB(\alpha)$ tree (because all $BB(\alpha)$ trees have $\Theta(\log N)$ heights);

II) Also, operations that involve inserting or deleting a leaf in a $BB(\alpha)$ tree can be executed in $\Theta(\log N)$ worst-case time. (The importance of $Alg(\alpha)$'s insertion and deletion operations is that they will assure that the relevant tree will continue to be a $BB(\alpha)$ tree after those operations are completed.)

This chapter's super-B-tree algorithm will be a generalization of the Nievergelt-Reingold algorithm. Consequently, the next several paragraphs will contain a much more detailed description of their $Alg(\alpha)$ procedure.

The intuitive idea behind this procedure is quite simple. Nievergelt and

Reingold based their new algorithm on the observation that it is fairly easy to rectify the value of $p(v)$ if the insertion or deletion of a given leaf causes this number to move outside the required $[\alpha, 1-\alpha]$ range.

In most such cases, Nievergelt and Reingold will use one of the four transformations that are shown in Diagrams 3. 2. D through 3. 2. G to force the value of $p(v)$ to return back into the $[\alpha, 1-\alpha]$ interval. These four possible transformations will be henceforth called the R_1, R_2, R_1' and R_2' rotations. As the reader examines the relevant diagrams, he should bear in mind that the circles in these diagrams represent nodes (such as v, v_A, v_B, etc.) and that the triangles represent subtrees (such as T_1, T_2, T_3, etc.). Also the reader should take note of the fact that the R_1 and R_2 rotations (in Diagrams 3. 2. D and 3. 2. E) are mirror images of the R_1' and R_2' rotations (that are shown in Diagrams 3. 2. F and 3. 2. G). This is because the former rotations will be used when weight must be <u>shifted leftward to decrease</u> the value of $p(v)$, and the latter rotations when weight must be <u>moved rightward to increase</u> the value of $p(v)$. Finally, it should be stated that my recommendation is that the reader should only briefly examine the below four rotation diagrams because a more detailed description of their significance will be given in the next several paragraphs:

Diagram 3. 2. D. The R_1 "single" rotation:

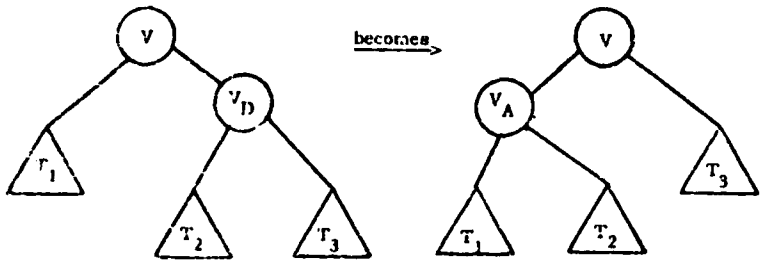


Diagram 3. 2. E. The R_2 "double rotation:

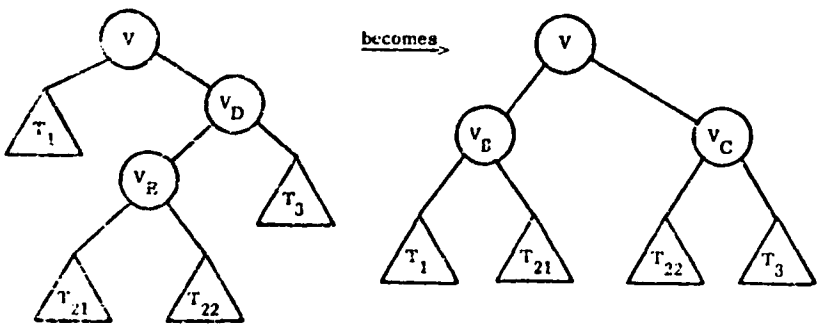


Diagram 3. 2. F. The R_1' "single" rotation:

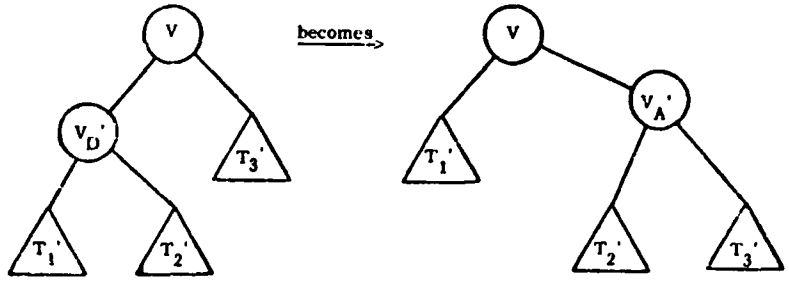
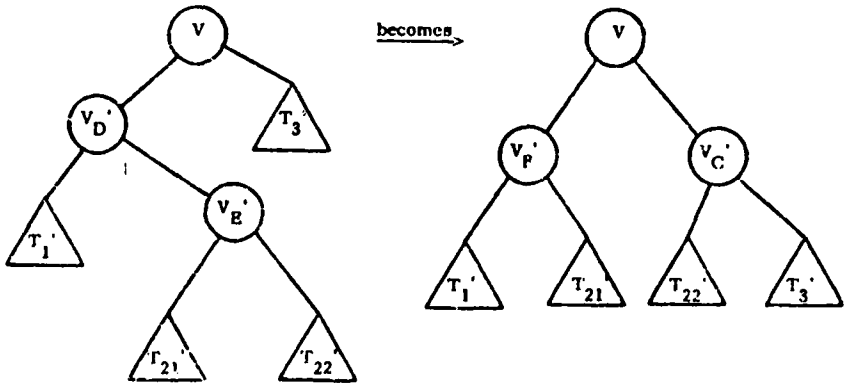


Diagram 3. 2. G. The R_2' "double" rotation:



The discussion of the preceeding four rotations will be divided into two parts. The first part will describe the special Nievergelt-Reingold subroutine that is used to determine which one of the R_1, R_2, R_1' or R_2' rotations should be employed to force the value of $p(v)$ to return back into the required $[\alpha, 1-\alpha]$ range. This subroutine will be called NR-DETERMINE(α), and it will be described in part 3.2.H of this section. The second part of this section will describe the mainline of the Nievergelt-Reingold Alg(α) procedure. The discussion of this topic will be given in part 3.2.I of this section. That discussion will explain how the Alg(α) mainline will employ the NR-DETERMINE(α) subroutine to help it attain its important $\Theta(\log N)$ runtime.

One final prefactory remark should be made before a formal description is given to Nievergelt's and Reingold's NR-DETERMINE(α) subroutine. In fairness to the authors, it should be stated that they did not assign any special name to this procedure. The name of "NR-DETERMINE" is thus used in this thesis but not in Nievergelt and Reingold's original paper. Our usage of this name is due to the fact that several parts of this chapter will refer to that special part of the Nievergelt-Reingold algorithm which will be called "NR-DETERMINE(α)" in this thesis.

The NR-DETERMINE(α) component of the Nievergelt-Reingold algorithm is described below:

<u>Subroutine 3.2.H.</u> The NR-DETERMINE(α) subroutine will be invoked by the mainline of the Alg(α) procedure when a single insertion of deletion operation causes the $p(v)$ value of a node v to move outside the $[\alpha, 1-\alpha]$ range. The purpose of the NR-DETERMINE subroutine will be to determine which one of the R_1, R_2, R_1' or R_2' rotation should be used to help $p(v)$ regain a value that belongs to the $[\alpha, 1-\alpha]$ interval. The general rules that Nievergelt and Reingold recommended were that:

A) If $p(v)$ is less than or equal to α then either the R_1 or R_2 rotations should be used to shift weight <u>leftward</u> through node v. The R_1 rotation will be used when equation 3 is satisfied, and the R_2 rotation otherwise:

3) $$p(v_D) \le \frac{1-2\alpha}{1-\alpha} \quad .$$

B) If $p(v)$ is greater than or equal to $1-\alpha$ then one of the R_1' or R_2' rotations should be used to shift weight <u>rightward</u> through node v. The R_1' rotation will be used when equation 4 is satisfied, and the R_2' rotation should be employed otherwise. It should also be stated that this case and equation 4 are the mirror image analogs of Case A and equation 3.

4) $$p(v_D') \ge 1 - \frac{1-2\alpha}{1-\alpha} \quad .$$

One final remark should be made about the NR-DETERMINE(α) subroutine. It will be assumed in this thesis that the NR-DETERMINE(α) subroutine

only recommends a rotation and that the actual physical implementation of this rotation will be done by the mainline program that made the subroutine-call to NR-DETERMINE(α). The reason for this assumption is that it will make it easier to explain the relationship between the NR-DETERMINE(α) subroutine and the super-B-tree mainline.

The detailed discussion of the super-B-tree algorithm will take place in section 3.4 of this thesis. The immediate goal of this section will be to explain how Nievergelt and Reingold's Alg(α) mainline will employ their NR-DETERMINE(α) subroutine. The formal description of their Alg(α) mainline is given below:

Algorithm 3.2.1. The Alg(α) procedure will be defined for any value of α that satisfies the inequality of:

5) $$0 < \alpha \leq 1 - \sqrt{2}/2 \quad .$$

Under these circumstances, the Alg(α) procedure will be designed to insert or delete any leaf in a BB(α) tree which is indicated by an initial command that is given by the user. The Alg(α) procedure will perform these operations in a manner which assures that:

I) all possible insertion and deletion commands can be executed in $\Theta(\log N)$ worst-case runtime;

II) the modified tree produced by these commands will also be a BB(α) tree.

The specific Alg(α) procedure that has been recommended by Nievergelt and Reingold will consist of the following three general types of steps:

1) The first step will take the Key of the leaf which is specified by the user's insertion (or deletion) command, and it will perform a simple topdown tree search in pursuit of that location in our B-tree where this Key should be stored.

2) The second step will perform that leaf-insertion (or leaf-deletion) operation at this location which is suggested by the user's initial (or deletion) command.

3) The third step will be designed to "rebalance" those interior nodes, v, whose p(v) values were caused to move outside the $[\alpha, 1-\alpha]$ range by the previous step. This step will take advantage of the fact that all the specified interior nodes will be ancestors of that leaf which the second step inserted (or deleted). The procedure in this step will perform a bottom up tree walk to find these ancestor nodes. A subroutine-call will be made to the NR-DETERMINE(α) procedure whenever this step finds an ancestor node whose p(v) value lies outside the required $[\alpha, 1-\alpha]$ interval. The NR-DETERMINE(α) subroutine will

be asked to decide which one of the R_1, R_2, R_1' or R_2' rotation should be used to force $p(v)$ to return back into the required $[\alpha, 1-\alpha]$ range. This step will physically perform those rotations which are specified by the NR-DETERMINE(α) subroutine. By the end of this step, nodes similar to v will be sufficiently "rebalanced" to insure that all the $p(v)$ values once again belong to the $[\alpha, 1-\alpha]$ interval.

A more detailed description of the preceeding Alg(α) procedure can of course be found in Nievergelt's and Reingold's original paper (NR-74). It must be once again stated that the significance of Alg(α) is that it enables Nievergelt and Reingold to define a new type of B-tree algorithm that has $\Theta(\log N)$ search, insertion, and deletion times.

Remark 3.2.1. The preceeding paragraph offered a fairly accurate (although somewhat simplified) description of the Nievergelt-Reingold algorithm. It should be stated that the preceeding Nievergelt-Reingold algorithm did contain one (extremely) minor error. This error is due to the unusual behavior of those nodes that have a small number of descendants. The basically trivial special case of those nodes that have less than α^{-2} leaf descendants was overlooked in NR-74. The Alg(α) procedure will occasionally use an improper rebalancing rotation when a node has less than α^{-2} leaf descendants (and α additionally is a small number). This error is due to the fact that

the R_1, R_2, R_1' and R_2' can be guaranteed to properly "rebalance" a node v only when this node has at least α^{-2} leaf descendants. This mistake in the Alg(α) procedure is unimportant because it is easy to correct. The Alg(α) procedure will be corrected if a new module is added to this algorithm for the handling of nodes that have less than α^{-2} leaf descendants. When such nodes need rebalancing, this module should use a special procedure to assure that these nodes and their descendants have left and right sons whose weight differ by no more than 1. It is trivial to verify that such a new module will not impair Nievergelt's and Reingold's basic result of an $\Theta(\log N)$ worst-case runtime (because the new module will only be applied to small subtrees that require little runtime). A more detailed discussion of Nievergelt's and Reingold's minor error will not take place in this thesis because

i) the error is trivial to correct

ii) it will not be directly related to this chapter's main proofs (which are given in section 3.5 and 3.6),

iii) and because Nievergelt's and Reingold's minor omission does not detract from the basic importance of the new concepts that they introduced.

There will be several generalizations of the Nievergelt-Reingold Alg(α) procedure that will be discussed during the development of the super-B-tree algorithm. These generalized algorithms will make subroutine-calls to a procedure which is a generalized version of Nievergelt and Reingold's

NR-DETERMINE(α) subroutine. This generalized subroutine will be called DETERMINE(α, K), and the next several paragraphs will be devoted to giving a detailed description to this subroutine.

The objective of the DETERMINE(α, K) subroutine will be very similar to that of the previous NR-DETERMINE(α) subroutine. Both of these algorithms will be designed to determine which one of the R_1, R_2, R_1' or R_2' rotations should be applied to a violating node whose $p(v)$ value has moved outside the required $[\alpha, 1-\alpha]$ range. The goal of both of these procedures will be to find that transformation which forces $p(v)$ to return back into the required interval. In all aspects but one, these two subroutines will use identical procedures to make this determination. The single distinction between these procedures will be that the DETERMINE(α, K) subroutine will be defined to employ equations that are slightly more complicated than equations 3 and 4 of the NR-DETERMINE(α) subroutine. The new equations of DETERMINE(α, K) will differ from their predecessors by replacing equation 3's and 4's "2α" terms with more general "$K\alpha$" terms. This change will be important because it will cause the DETERMINE(α, K) and NR-DETERMINE(α) procedures to sometimes make different recommendations about which one of the R_1, R_2, R_1' or R_2' rotations should be applied.

A much more detailed description of the DETERMINE(α, K) subroutine and its intended applications will be given in the next several paragraphs.

That discussion will begin with a formal description of this subroutine. My recommendation is that the reader should only skim this formal subroutine description because the DETERMINE(α, K) procedure will be identical to NR-DETERMINE(α) in all respects except for the single difference that was mentioned in the last paragraph. There are two other initial comments that should be made before the reader examines the DETERMINE(α, K) procedure. The first is that the DETERMINE(α, K) subroutine will become identical to NR-DETERMINE(α) when K is assigned the value of 2. Secondly, the reader should keep in mind that the DETERMINE(α, K) subroutine is being introduced in this thesis because the super-B-tree algorithm will need it.

Subroutine 3.2.K. The DETERMINE(α, K) procedure will be invoked when the insertion or deletion of a leaf causes a $p(v)$ value to move outside the required $[\alpha, 1-\alpha]$ range. The purpose of this subroutine will be to determine which one of the R_1, R_2, R_1' or R_2' rotations should be used to help $p(v)$ regain a value that belongs to the $[\alpha, 1-\alpha]$ interval. The general rules of this algorithm are a minor generalization of NR-DETERMINE(α), and they are given below:

A) If $p(v)$ is less than or equal to α then either the R_1 or R_2 rotations should be used to shift weight leftward through node v. The R_1 rotation will be used when equation 6 is

satisfied, and the R_2 rotation otherwise:

6) $$p(v_D) \leq \frac{1 - K\alpha}{1 - \alpha} \quad .$$

B) If $p(v)$ is greater than or equal to $1-\alpha$ then one of the R_1' or R_2' rotations should be used to shift weight rightward through node v. The R_1' rotation will be used when equation 7 is satisfied, and the R_2' rotation should be employed otherwise. It should also be stated that this case and equation 7 are the mirror image analogs of Case A and equation 6:

7) $$p(v_D') \geq 1 - \frac{1 - K\alpha}{1 - \alpha} \quad .$$

Finally, it must be once again reiterated that neither the NR-DETERMINE(α) nor the DETERMINE(α, K) subroutines will physically perform that rotation which they recommend to the calling mainline program. Instead, the responsibility for physically implementing these rotations will be delegated to the mainline programs. The reason for this stipulation will become apparent during section 3.4 through 3.6 of this thesis. There we will see that the recommendations of the DETERMINE(α, K) subroutine will sometimes be ignored.

There will be two mainline algorithms that will make subroutine-calls to DETERMINE(α, K) in this thesis. These will be the Alg(α, β, K, J) and Alg(α, K) mainlines. The former algorithm is so complicated that its

fundamental characteristics cannot even be briefly summarized. The latter algorithm will be a simplified version of the former procedure whose principle advantages is that it is much easier to describe. The rest of this section will be devoted to describing the Alg(α, K) procedure and to explaining its advantages over the more traditional B-tree algorithms. By the end of this section, the reader should begin to understand the connection between the Nievergelt-Reingold algorithm and super-B-trees.

Throughout this chapter, Q(K) will denote the following quadratic

8) $$Q(K) = 1 - \frac{\sqrt{K^2 - K}}{K} \quad .$$

Also, in most of the remainder of this chapter, it will be assumed that K and α satisfy the following two equations:

9) $$K \geq 2$$

10) $$0 < \alpha \leq Q(K) \quad .$$

The Alg(α, K) procedure will be defined for any values of α and K that satisfy the preceeding two equations. When defined, this procedure will have a goal that is basically identical to the goals of Nievergelt and Reingold's Alg(α) procedure. The purpose of the Alg(α, K) procedure will thus be to manipulate BB(α) in a manner that assures that any leaf can be inserted or deleted in $\Theta(\log N)$ time and which additionally makes certain that this tree will continue to be a BB(α) tree at the time when these insertion and deletion

operations are completed.

The specific procedure that is utilized by the $Alg(\alpha, K)$ procedure will be virtually identical to that of $Alg(\alpha)$. The only difference between these procedures will be that $Alg(\alpha, K)$ will make a subroutine-call to DETERMINE(α, K) in those places where $Alg(\alpha)$ made subroutine-calls to the less general NR-DETERMINE(α) subroutine. A formal description of the $Alg(\alpha, K)$ procedure will not be included in this thesis because the earlier formal description of the $Alg(\alpha)$ procedure together with the preceeding sentence offers a sufficiently precise description of this $Alg(\alpha, K)$.

It should be stated that the correctness and $\Theta(\log N)$ worst-case runtime of the $Alg(\alpha, K)$ procedure can be proven with techniques that are virtually identical to those that Nievergelt and Reingold applied to $Alg(\alpha)$. This is because the Nievergelt-Reingold $Alg(\alpha)$ procedure should be regarded as the specialized version of $Alg(\alpha, K)$ that results when $K = 2$. It thus follows that the Nievergelt-Reingold proofs can be trivially generalized if 2 is replaced by K. (For example $Q(K)$ in equation 10 becomes equation 5's number of $1 - \frac{\sqrt{2}}{2}$ when $K = 2$.)

This section will not contain any further detailed proof about the correctness or $\Theta(\log N)$ runtime of the $Alg(\alpha, K)$ procedure for two reasons. The first is that later parts of this chapter will

(formally) prove a much more powerful theorem about a further generalized $Alg(\alpha, \beta, K, J)$ procedure. The second reason a formal proof about $Alg(\alpha, K)$ has been omitted is that all aspects of this proof and the $Alg(\alpha, K)$ procedure are absolutely trivial generalizations of the previous results which Nievergelt and Reingold obtained for the $Alg(\alpha)$ procedure.

One final prefactory remark should be made about the $Alg(\alpha, K)$ procedure before we turn our attention to the topic of SDS fields. It should be stated that the $Alg(\alpha, K)$ procedure is such a minor generalization of Nievergeld and Reingold's previous work that it would not even warrant mentioning in the context of traditional B-tree applications. The reason for our interest in this subject is due to the rather unusual nature of the B-tree applications that will be considered in this chapter. These B-trees will differ from traditional B-trees insofar as they will possess SDS fields. The implications of SDS fields will be discussed in the rest of this section. That discussion will show how the $Alg(\alpha, K)$ procedure becomes very important in the context of SDS fields.

During the discussion of this topic, it will be necessary to use new notation and definitions. The relevant terminology is introduced in the next several paragraphs.

In this chapter, the small letter symbol of "c_i" will denote some command given by the user to insert or delete a specified record. Also, the

capital letter symbol of "C" will be used in this chapter. The latter symbol will denote a sequence of individual insertion and deletion commands.

It often will be useful in this chapter to determine how many commands belong to sequence C. The symbol of $|C|$ will therefore denote the number of commands in this sequence. The following definitons will use the preceeding terminology to define two important criteria for measuring the runtime of tree manipulating algorithms.

Definition 3.2.L. Let T denote a tree and A denote an algorithm that manipulates the nodes in this tree. Let us assume that tree T initially contains no leaves and that the user subsequently gives some sequence of insertion and deletion commands. Let C denote this sequence of commands. The algorithm of A will be said to have a "conservatively estimated runtime" (CERT) equal to r if $|C|r$ represent the maximum possible amount of runtime that any sequence, C, can force algorithm A to consume.

Comment: The significance of the above defined CERT runtime can be best appreciated if it is compared to the concepts of "worst-case" and "stochastically expected" runtimes. Note that the concept of CERT runtime is weaker than worst-case runtime because CERT runtime allows an individual insertion or deletion command to consume more than r time. Also, note that, CERT runtime is stronger than stochastically expected runtime because the latter concept would make it possible for sequence C to consume more than $|C|r$

time (if certain improbable circumstances prevailed). The concept of CERT runtime thus refers to a notion that is intermediate between the concepts of worst-case and stochastically expected runtime. We will now define a second concept called "SCERT" runtime. This concept will be stronger than CERT runtime but still weaker than "worst-case" runtime.

Definition 3.2.M. Let T and A once again denote a tree and an associated algorithm that manipulates this tree. Let N_O denote the number of leaves that initially belongs to this tree, and let C denote a sequence of insertion and deletion commands which is subsequently given to modify this tree. Algorithm A will be said to have a "strictly conservatively estimated runtime" (SCERT) equal to r if $(|C| + N_O)r$ represents the maximum possible amount of runtime that any sequence C and tree T can force algorithm A to consume.

Comment: The concept of SCERT runtime is somewhat more powerful than CERT runtime. The reason for this can be best understood if the example is considered where the tree T initially is empty and where two sequences of commands (denoted as C_1 and C_2) are subsequently given. Let N_O denote the size of this tree after the C_1 sequence of commands is given. Let us further assume that $|C_2| << |C_1|$ and $N_O << |C_1|$. In this case, it is fairly easy to construct examples where a tree manipulating algorithm could have a CERT runtime that equals $\log N$ and a SCERT runtime that is

an arbitrarily large function of N. It thus follows that SCERT runtime is stronger than the concept of CERT runtime.

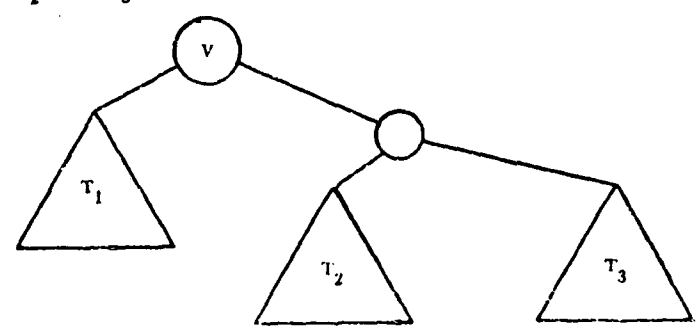
Let A denote some B-tree algorithm that is not initially designed to manipulate SDS fields (such as the "AVL," "2-3," "multi-way" and "bounded balance" algorithms). Let T denote a tree whose nodes contain all the fields required by algorithm A and whose nodes additionally contain pointers to SDS structures. The SDS generalization of algorithm A will be defined to be that tree manipulating procedure that possesses all the modules that are associated with algorithm A plus the "obvious" additional module that is needed to properly maintain T's SDS fields. (This additional module will use a very simple procedure for processing SDS field. It will simply spend $|\cdot w$ units of runtime updating node v's SDS structure whenever this node gains or loses $|$ descendant leaves.)

Note that the runtimes of the SDS generalizations of the AVL, 2-3, multi-way, and bounded balance algorithms were briefly mentioned in the introductory part of this chapter. That section indicated that worst-case examples could be constructed where a single insertion or deletion command could require $\Theta(wN)$ time. This estimate of the worst-case runtime left open the possibility that the CERT or SCERT measurements of the runtimes of these algorithms might have a more amenable value. The next two theorems will show that such a result mostly does not hold. The CERT and SCERT runtimes of the

AVL, 2-3, multi-way and $Alg(\alpha)$ procedures will all be shown to have very inefficient $\Theta(wN)$ magnitudes. Only, the $Alg(\alpha, K)$ procedure will be shown to have a better runtime. Theorem 3.2.S of this section will state that $Alg(\alpha, K)$ will have an $\Theta(w \log N)$ CERT and SCERT runtime (when K is assigned the proper initial value). The purpose of all the theorems of this section will be to explain the many difficulties which section 3.4's super-B-tree algorithms will be designed to overcome.

THEOREM 3.2.N. Let α denote a standard Nievergelt-Reingold real number that is less than or equal to $1 - \sqrt{2}/2$ and let $Alg(\alpha)$ denote the standard Nievergelt-Reingold algorithm that manipulates $BB(\alpha)$ trees. The SDS generalization of $Alg(\alpha)$ will have CERT and SCERT runtimes that have $\Theta(wN)$ magnitudes.

Proof: Let T denote a $BB(\alpha)$ tree which contains the three subtrees of T_1, T_2 and T_3 as shown in the diagram below:



Let us further assume that these three subtrees contain respectively αN, $(1-2\alpha)N$, and αN leaves. Let $c_1\ c_2\ c_3\ c_4$ denote a sequence of four commands such that

1) Command c_1 inserts a leaf into the T_3 subtree;

2) Command c_2 deletes the same leaf;

3) Command c_3 inserts a leaf into the T_1 tree;

4) Command c_4 deletes the same leaf.

It is easy to verify that the previous four commands will cause $Alg(\alpha)$ to move the T_2 subtree from the right side of node v to its left side and then back to the initial position. Note that these movements will force $Alg(\alpha)$ to consume $\Theta(wN)$ time (because the SDS structures must be adjusted whenever T_2 moves). Also note that the cycle of the previous 4 commands can be repeated any number of times. Each repetition will consume $\Theta(wN)$ additional time. Such reasoning immediately implies that $Alg(\alpha)$ will have $\Theta(wN)$ CERT and SCERT runtimes.

Q. E. D.

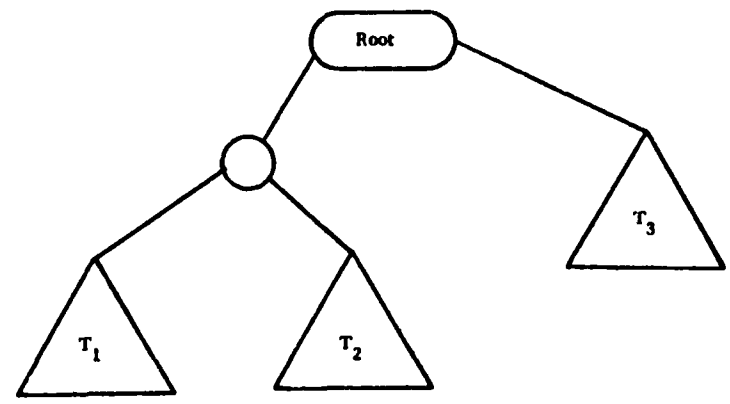
<u>THEOREM 3.2.O.</u> The CERT and SCERT runtimes of the SDS versions of the AVL, 2-3 and multiway algorithms will have at least an $\Theta(wN)$ magnitude.

Some readers will probably prefer to omit the rather lengthy proof of this theorem. This is because AVL, 2-3, and multi-way algorithms are not directly related to the next section's super-B-tree algorithm. Thus,

Theorem 3.2.O has only been included in this thesis for the sake of completeness. It allows us to <u>compare</u> the relative runtime merits (and) demerits) of the various different types of B-tree algorithms.

The proof of Theorem 3.2.O will presume that the reader is previously familiar with the AVL, 2-3 and multi-way algorithms. (These three procedures were discussed in AVL-62, AHU-74, and Kneuth-73.) The proof will be divided into three parts where each of these three algorithms are separately discussed:

<u>Proof for AVL trees:</u> A binary tree of height h will be said to have maximal size if it contains precisely 2^h leaves. Let $T_1\ T_2$ and T_3 denote three such maximal sized trees of height h. Let T denote that AVL tree which is shown in the diagram below:



Let $c_1 c_2 c_3 c_4$ denote a sequence of four data-modification commands such that

i) Command c_1 causes a leaf to be added to subtree T_1;

ii) Command c_2 removes the cited leaf;

iii) Command c_3 causes a leaf to be added to subtree T_3;

iv) Command c_4 removes the cited leaf.

It is easy to verify that the previous four commands will cause the AVL algorithm to move T_2 from the left side of tree T to the right side and then subsequently back to the initial position. Note that this cycle of four commands will require $\Theta(wN)$ time to manipulate the SDS fields. Furthermore, this cycle can be repeated any number of times. Each repetition will thus consume an additional $\Theta(wN)$ time. The preceeding implies that the SDS generalization of the AVL algorithm will have at least an $\Theta(wN)$ CERT and SCERT runtime.

Q. E. D.

<u>Proof for the case of 2-3 Trees:</u> Consider a 2-3 tree where every ancestor of leaf L has 3 sons. Consider a sequence of two commands where:

i) Command c_1 causes L to gain a new brother;

ii) Command c_2 removes the cited record.

Note that these two commands will force the SDS generalization of the 2-3 algorithm to consume $\Theta(wN)$ time. Also, note that the 2-3 tree will be

in the same state after the execution of these two commands as it was before. Each repetition of a cycle of these two commands will thus consume an additional $\Theta(wN)$ time. Such reasoning implies that the SDS generalization of 2-3 trees will have at least an $\Theta(wN)$ CERT and SCERT runtimes.

Q. E. D.

<u>Proof for multi-way trees:</u> Note that multi-way trees are the generalization of 2-3 tree for the case when each ancestor has between m and $2m - 1$ sons. The preceeding proof for 2-3 trees can consequently be easily modified to show that multi-way trees also have at least $\Theta(wN)$ CERT and SCERT runtimes.

Q. E. D.

<u>Remark 3.2.F:</u> Some readers may perhaps question whether it is fair to judge the SDS generalizations of the AVL, 2-3, multi-way, and bounded balance tree algorithms according to their CERT and SCERT runtimes. These readers may point out that all four of these SDS generalizations will be (stochastically) expected to have an $\Theta(w \log N)$ runtime (if a uniform probability distribution is presumed). There are two reasons why the preceeding CERT and SCERT runtimes are important despite this $\Theta(w \log N)$ statistically expected runtime. These are that:

i) In many practical applications, the relevant database will be generated in a much more complicated manner than is suggested by a more uniform probability distribution. Realistic examples

can thus be constructed of time-dependent non-uniform probability distributions whose expected runtimes have greater than $\Theta(w \log N)$ magnitude;

2) Also, many users will be dissatisfied with $\Theta(wN)$ CERT and SCERT runtimes of the previous four algorithms because those runtimes do not fully preclude the possibility that K consecutive commands could consume $\Theta(KNw)$ time (for any value of the integer of K).

The remainder of this section will be devoted to proving that the $Alg(\alpha, K)$ generalization of the bounded balance algorithm will manipulate SDS field in a much better runtime than did the previous four B-tree procedures. The discussion in this section will begin with a preliminary definition and lemma. These initial concepts will help analyze the runtime of the $Alg(\alpha, K)$ procedure.

Definition 3.2.Q. Let T denote some tree. The leaf depth of that tree will be defined to be the sum of the depths of its leaves.

The next lemma will compare the concept of leaf-depth to the amount of runtime that is consumed by the $Alg(\alpha, K)$ procedure. This lemma will later help prove this section's main theorem.

LEMMA 3.2.R. Let K and α denote the standard numbers which are used to generate an $Alg(\alpha, K)$ procedure. Note that the SDS generalization of this procedure will be required to adjust the SDS fields of the interior nodes whenever the "rebalancing" step of this procedure performs one of those transformations that were previously outlined in Diagram 3.2.D through 3.2.G. If $K > 2$ then the amount of time that the $Alg(\alpha, K)$ procedure will need to make such adjustments in the SDS fields will always be proportional to the amount of leaf-depth that is lost during these rebalancing transformations. More specifically, the ratio of time spent adjusting the SDS fields over loss of leaf-depth (during rebalancing) will always be less than or equal to $\frac{2w}{(K-2)\alpha}$.

Proof: It is easiest to prove this lemma if the transformations in Diagrams 3.2.D through 3.2.G are re-examined. A consideration of all possible cases indicates that if N_v denotes the number of leaves which descend from v and if R denotes that one of the R_1, R_2, R_1' or R_2' rotation which is chosen by the DETERMINE(α, K) subroutine then rotation R will necessarily produce a loss of leaf-depth that is at least as large as $N_v(K-2)\alpha$. Also, rotation R will require no more $2N_v w$ time to build the new SDS structures (for any of the nodes of v_A, v_B, v_C, v_A', v_B', or v_C'). These two facts imply that the ratio of amount of time needed to adjust the SDS fields over loss of leaf-depth will always be less than or equal to $\frac{2w}{(K-2)\alpha}$.

Q.E.D.

THEOREM 3.2.S. Let K denote some number which is greater than 2 and α denote the standard Nievergelt-Reingold number that is less than or equal to $Q(K)$ (which was defined in equation 8). The SDS generalization of the $Alg(\alpha, K)$ procedure will possess $\Theta(w \log N)$ CERT and SCERT runtimes.

Proof: Note that the only part of the $Alg(\alpha, K)$ procedure which is capable of taking more than $\Theta(w \log N)$ time is its rebalancing step. Also, note that the previous lemma indicated that the runtime of the step is proportional to the loss of leaf-depth. The theorem will thus be proven if sufficient bounds are produced on the size of the leaf-depth of $BB(\alpha)$ trees.

Such bounds will be established in this paragraph. Let N denote the maximum size which tree T attains during command string C. Also, let N_O denote the initial size which this tree possesses before these commands are given. Note that Nievergelt and Reingold showed that all $BB(\alpha)$ trees have $\Theta(\log N)$ heights. It thus follows that

i) Tree T will begin with a leaf-depth that is no greater than $\Theta(N_O \log N_O)$.

ii) All the insertion commands in sequence C will increase the leaf-depth by no more than $\Theta(|C| \log N)$.

The previous two bounds on the size of the leaf-depths combined with

Lemma 3.2.R imply that $Alg(\alpha, K)$ must have an $\Theta(w \log N)$ CERT and SCERT runtime.

Q. E. D.

Comment: It is interesting to re-examine Nievergelt's and Reingold's original $Alg(\alpha)$ procedure in light of the previous two propositions. Note that the previous two theorems only hold when $K > 2$ and that $Alg(\alpha)$ was the special version of $Alg(\alpha, K)$ that resulted when K equaled 2. It is for this reason that the SDS generalization of $Alg(\alpha, K)$ was more efficient than the SDS generalization of $Alg(\alpha)$. The implications of these facts will be utilized in the remainder of this chapter. The "super-B-tree" algorithm will be a further generalization of $Alg(\alpha, K)$ that is designed to convert Theorem 3.2.S's $\Theta(w \log N)$ CERT and SCERT runtime into an $\Theta(w \log N)$ worst-case runtime.

3.3 An Intuitive Description of "Super-B-Trees"

The rest of this chapter will be devoted to the study of the super-B-tree algorithm. This procedure will be somewhat more powerful than $Alg(\alpha, K)$ because it will manipulate SDS fields in $\Theta(w \log N)$ WORST-CASE runtime.

The general nature of the forthcoming discussion can be best explained if the example is considered where the user must decide whether a sequence of N commands should be executed with either a first algorithm that spends 3 units of time per command or with a second algorithm that executes one of these commands in N units of runtime and which executes the other commands in one unit of time per instruction. In most cases, users will prefer the first algorithm over the second because the first will operate in a more predictable and consistent amount of runtime. Consistency is important in most database applications because unexpectedly expensive commands can paralyze these systems. Thus, it is usually safer for a database management system to employ a consistent procedure that has a slightly higher average runtime instead of an unpredictable procedure that occasionally consumes a prohibitive amount of runtime.

During this thesis, the reader should bear in mind the importance of consistent performance. Many of the complicated aspects of this thesis are due to the requirements of such consistency. Often in this thesis, it will be necessary to attain the required level of consistency by performing a trade-off that

partially sacrifices stochastically expected runtime for the sake of consistency. An example of one possible such trade-off was given in the last paragraph.

The super-B-tree procedure will be another example of an algorithm that employs the trade-off between consistency and stochastically expected runtime. The remainder of this chapter will show that the super-B-tree procedure will attain its $\Theta(w \log N)$ worst-case time at the expense of (stochastically) expected runtime. This trade-off will cause super-B-tree procedure to differ from $Alg(\alpha, K)$ by having an improved worst-case runtime and slightly worsened CERT, SCERT, and stochastically expected runtimes. This trade-off will be shown to be very useful because it will produce a great improvement in worst-case runtime at the expense of only a minor worsening of the runtime coefficients of CERT, SCERT and stochastically expected runtimes.

A formal description of the super-B-tree algorithm will be given in the next three sections of this chapter. The remainder of this section will be devoted to giving a more intuitive description of this procedure. This discussion will begin with the following very important definition:

Definition 3.3.A. Let R denote any one of the R_1, R_2, R_1' or R_2' transformations that were shown in Diagrams 3.2.D through 3.2.G (of the previous section). Note that these transformations will cause certain older nodes (such as v_D, v_E, v_D' and v_E') to be replaced by newer nodes

(such as v_A, v_B, v_C, v_A', v_B' and v_C'). Those older nodes which transformation R shall destroy will be called its "removal nodes", and the new nodes that it shall create will be called its "renewal nodes".

Example 3.3.B An examination of Diagrams 3.2.D through 3.2.G indicates that

i) Transformation R_1 has a removal node of v_D and a renewal node of v_A.

ii) Transformation R_2 has removal nodes of v_D and v_B and renewal nodes of v_B and v_C.

iii) Transformation R_1' has a removal node of v_D' and a renewal node of v_A'.

iv) Transformation R_2' has removal nodes of v_D' and v_E' and renewal nodes of v_B' and v_C'.

The next two sections of this chapter will explain how the super-B-tree algorithm will use the R_1, R_2, R_1' and R_2' rotations to "rebalance" its B-trees. In most respects, these rebalancing operations of the super-B-tree algorithm will be very similar to the last section's Alg(α, K). The prime difference between these two algorithms can be understood if we take a closer look at how these rebalancing transformations will affect their SDS fields.

The significant aspect about the R_1, R_2, R_1' and R_2' rotations is that they will require that the old SDS fields of their removal nodes be

replaced by the SDS fields of their renewal nodes. These replacement operations will be potentially expensive, and they are the main reason that the Alg(α, K) procedure possessed an $\Theta(wN)$ worst-case runtime. The key goal of the super-B-tree algorithm will be to find an alternate way of performing these replacement operations that is sufficiently efficient to support an $\Theta(w \log N)$ worst-case runtime.

A detailed description of the super-B-tree algorithm will be given in the next section. This algorithm will differ from Alg(α, K) primarily because it will attempt to anticipate those new SDS fields that it is likely to need in the future. Unlike Alg(α, K), the super-B-tree procedure will not simply postpone the task of constructing a new SDS field until that time when a R_1, R_2, R_1' or R_2' rotation needs such a field. Instead, the super-B-tree algorithm will gradually construct these new SDS fields in anticipation of the future needs of these rebalancing transformations. The purpose of such a preparatory procedure will be to gather sufficient preliminary information to insure that the R_1, R_2, R_1' and R_2' rotations can always be executed in $\Theta(1)$ runtime. The main theorem in this chapter will state that such techniques will enable the super-B-tree algorithm to attain its proclaimed $\Theta(w \log N)$ worst-case runtime.

It should be stated that section 3.6's super-B-tree theorem will be very closely related to Theorem 3.2.S (of the last section). That theorem stated that the Alg(α, K) procedure would have an $\Theta(w \log N)$ CERT and SCERT

runtime. The super-B-tree algorithm will basically be a revised version of Alg(α, K) that is designed to convert the previously discussed Θ(w log N) CERT and SCERT runtimes into an Θ(w log N) worst-case runtime. The super-B-tree algorithm will attain this runtime by constructing its "anticipated" future-needed SDS field over an extended period of time while the user is giving a large number of insertion and deletion commands. This construction procedure will be carefully designed to insure that only a small part of this construction task takes place during any single insertion or deletion command. The purpose of this "gradual" construction will be to make certain that only the runtime coefficients of insertion and deletion operations are increased during the period when the anticipated SDS fields are gradually constructed. A more detailed description of the "gradual" construction procedure will be given in the next three sections of this chapter. There we will see that this gradual construction procedure enables the super-B-tree algorithm to attain its Θ(w log N) worst-case time at the expense of only a modest increase in the runtime coefficient of the stochastically expected runtime.

It should be stated that the discussion in the remainder of this chapter will be quite complicated. This is because the super-B-tree algorithm must employ several special modules for several special cases. Some readers may consequently prefer to omit the remaining sections of this chapter on their first reading of this thesis. The thesis subject matter has been organized so that the

reader can omit those sections provided he accepts without proof my claim that the super-B-tree algorithm will operate in Θ(w log N) worst-case time.

3.4 The Mainline of the Alg(α, β, K, j) Procedure

The rest of this chapter will be devoted to the formal description of the super-B-tree algorithm. The discussion of this topic will be divided into three parts. The formal definition of the mainline of the super-B-tree procedure will be given in this section. Sections 3.5 and 3.6 will subsequently prove the correctness and $\Theta(w \log N)$ worst-case runtime of this procedure.

There are several preliminary definitions which must be introduced before the super-B-tree mainline can be formally described. These definitions will be introduced in the first half of this section.

Definition 3.4.A. Let v denote a node that is currently existing in our B-tree, and let R denote any one of the R_1, R_2, R_1' or R_2' rotation transformations (that were previously illustrated in Diagrams 3.2.D through 3.2.G). Under these circumstances, the symbol of ANTICIP(v, R) will denote the set of renewal nodes which are produced when rotation R is applied to that subtree of our super-B-tree which has a root of v. Also, the union of the four sets of ANTICIP(v, R_1), ANTICIP(v, R_2), ANTICIP(v, R_1') and ANTICIP(v, R_2') will be called "v's set of anticipated nodes" and will be denoted as ANTICIP(v).

Example 3.4.B. Note that Example 3.3.B indicated that there were six renewals which the R_1, R_2, R_1' and R_2' rotations would produce when they are

applied to the subtree whose root is v. (Those six nodes were previously denoted as v_A, v_B, v_C, $v_{A'}$, $v_{B'}$, and $v_{C'}$.) These six nodes will thus constitute the membership of the ANTICIP(v) set.

Definition 3.4.C. During this section, the nodes that belong to sets similar to ANTICIP(v) will be called "anticipated nodes." These "anticipated nodes" will represent future interior nodes that the super-B-tree algorithm is likely to soon be inserting into its tree data structure. It will be frequently necessary to distinguish the "anticipated" nodes from those nodes which are presently existing in the super-B-tree. The nodes that are currently existing in the super-B-tree will be called its "current nodes." Througiout this section, the symbol of v will denote an interior node that is either an "anticipated" or a "current" node.

There will be several other useful rules of notation that will be used in this section. The symbol of y will denote a leaf in a super-B-tree, the symbol of KEY(y) will denote the Key that is associated with this leaf, and the symbols of INF(v) and SUP(v) will denote the respective minimum and maximum Key-values that are associated with the subtree which has a root of v. The latter two symbols will have a slightly different definition for the cases of "current" and "anticipated" interior nodes. In the case of current nodes, INF(v) and SUP(v) will be defined to be equal to the respective minimum and maximum Key-values that are generated by the set of y-leaves which are

a descendant of v. In the case of anticipated nodes, INF(v) and SUP(v) will be defined to be equal to the minimum and maximum values that our algorithm "ANTICIPATES" would be associated with node v IF THAT NODE WERE to be placed into the super-B-tree. (Some readers may wish me to more fully explain precisely how the super-B-tree algorithm will calculate an INF(v) or SUP(v) value for an anticipated node. Such a calculation is quite easy to perform. Note that each "anticipated node" will be associated with a R_1, R_2, R_1' or R_2' rotation that produces this node. The INF and SUP values of this anticipated node will therefore be defined to be equal to be the respective minimum and maximum Key values of those leaves that would become a descendant of this anticipated node if the cited rotation were to insert this node into the super-B-tree.)

Example 3.4.D. Let us examine the ANTICIP(v, R_2) set and calculate the INF and SUP values that are associated with its anticipated nodes. The general nature of the R_2 rotation was shown in Diagram 3.2.E. (This diagram can be found on page 48.) Note that the cited diagram showed that the two renewal nodes associated with the R_2 rotation were v_B and v_C. In this example, we will calculate the INF and SUP values for these two members of the ANTICIP(v, R_2) set.

It is fairly easy to calculate these two INF and SUP values. Let us begin by examining node v_B. Note Diagram 3.2.E showed that all y-leaves

which are anticipated to descend from v_B will either belong to the T_1 or T_{21} subtrees. INF(v_B) will thus equal the smallest Key that is associated with this set of leaves, and SUP(v_B) will similarly equal its largest Key value. Also the values of INF(v_C) and SUP(v_C) can be presumed from Diagram 3.2.E to equal the respective smallest and largest Key-values that are associated with the leaves belonging to the T_{22} and T_3 subtrees.

During this section, the reader should bear in mind that all super-B-trees will have their leaves arranged by order of increasing Key value. This fact will imply that leaf y is a descendant of node v if and only if the following condition is satisfied

1) $$\text{INF}(v) \leq \text{KEY}(y) \leq \text{SUP}(v) \quad .$$

Definition 3.4.E. Let v denote a current or anticipated interior node in our super-B-tree. The symbol of SDS(v) will denote the SDS field that is associated with this node. There will be two types of SDS fields that will be discussed in this section. These are the "fully constructed" and "partially constructed" SDS fields. The former concept will refer to an SDS field that describes the full set of y-leaves that are a descendant of node v. This SDS field will, in other words, describe that set of y-leaves which satisfy the last paragraph's equation 1. The definition of the latter type of SDS field will require the existence of a number, TEMP(v) that satisfies the condition of:

2) $\quad INF(v) \leq TEMP(v) \leq SUP(v)$.

Under these circumstances, v's "partially constructed SDS field" will be defined to describe that set of y-leaves that satisfy the condition of:

3) $\quad INF(v) \leq KEY(y) \leq TEMP(v)$.

During the discussion of partially constructed SDS fields, the reader should bear in mind that equation 2 implies that equation 3 describes a subset of the set of y-leaves that satisfy equation 1. This fact means that the y-leaves in a partially constructed SDS field will necessarily be a subset of v's set of descendants. This is of course the reason that "partially constructed" SDS fields are so named. These "partially constructed" SDS fields will thus represent nothing other than a partial construction of a data structure that will eventually become a "fully constructed" SDS field.

One further remark should be made in reference to partially constructed SDS fields. For each node v that has a partially constructed SDS field, our super-B-trees will utilize a pointer that contains the address of that y-leaf whose Key value equals TEMP(v). This pointer will be henceforth denoted as PTEMP(v). The importance of PTEMP(v) will be explained later.

The formal definition of super-B-trees will be given in the next paragraph. That definition has been carefully designed to assure that all "current" interior nodes will have fully constructed SDS fields and that all

"anticipated" interior nodes will have either partially or fully constructed SDS fields.

Definition 3.4.F. A super-B-tree will be defined to be a tree whose leaves are arranged by order of increasing Key-value and which satisfies the following three important conditions:

A) Every interior node, v, of this tree must possess the standard fields which were used by the Nievergelt-Reingold "bounded balance" trees* plus thirteen additional fields. These thirteen additional fields will be

i) a pointer to v's SDS field

---

* There were four fields that Nievergelt and Reingold assigned to every interior node, v, of their "bounded balance" trees. These fields were two pointers to the left and right sons of v, a size field which indicated the number of v's descendants, and a KEY field whose defining characteristic was that all the leaves that were descendants of v's left son must have Key values that are smaller than this number and that the descendants of v's right son must similarly have larger Key values. The preceding four Nievergelt-Reingold fields will not play a major role in this chapter. This is because they do not exemplify the unique characteristics that distinguish super-B-trees from the standard Nievergelt-Reingold trees. Consequently, in the rest of this chapter, it will be implicitly assumed that the super-B-tree algorithm will manipulate the Nievergelt-Reingold fields in the more or less obvious manner that is suggested by the names of those fields.

ii) six pointers to the six SDS fields that are associated with the six members of the ANTICIP(v) set

iii) six PTEMP fields that contain the PTEMP pointers that are associated with the same six members of the ANTICIP(v) set. (Note that the definition of PTEMP was given two paragraphs earlier in this section.)

B) The second condition which super-B-trees must satisfy is that the SDS fields of all the "current" nodes in these trees must be fully constructed and that the SDS fields of the "anticipated" nodes may be either fully or partially constructed.

C) Finally, the leaves of a super-B-tree will be defined to have four fields. These four fields will be a Key, a "forward pointer" to the leaf that lies to the immediate right of the current leaf, a "back pointer" to the leaf that lies to the immediate left of the current leaf, and a fourth auxiliary field where the user can store any information he desires. (Note that the cited forward and back pointer will cause the leaves of a super-B-tree to form a list that has arranged these leaves by order of increasing Key value.)

Definition 3.4.G. Let S denote a set of leaves. Any super-B-tree data-structure which possesses this set of leaves will be called a representation of S.

Comment: It is easy to see that there are many different permissible super-B-tree representations of the same set S of leaves. This is because Definition 3.4.E allowed the anticipated SDS fields to either be fully or partially constructed and because the cited definition contained no specification on which particular internal tree structure should be used to arrange the interior nodes of a super-B-tree that represents the set S of leaves. The many possible representations of the same set S of leaves will be studied throughout this chapter. One of our goals will be to insure that this set S is give . an efficient super-B-tree representation.

The rest of this section will be devoted to describing the mainline of the super-B-tree algorithm as well as describing several of its subroutines and parameters. This discussion will begin with description of a subroutine that will be called the GCAS procedure.

The name of GCAS will be acronym for "Gradual Construction of Anticipated SDS fields." This name basically describes the purpose of the GCAS subroutine. The arguments of this subroutine will consist of an interior current node of v and an integer of J. The purpose of the GCAS procedure will be to insert J new leaves into each of the SDS fields of

those nodes which belong to the ANTICIP(v) set and which additionally satisfy the condition that their SDS fields are only "partially constructed." The net effect of several subroutine-calls to GCAS will be to cause the anticipated SDS fields to gain J new leaves on each occasion when this procedure is invoked until that time when this field becomes "fully constructed." Such a technique of gradually constructing the needed SDS fields over an extended period of time will be desirable because it will insure that no individual invocation of the GCAS procedure will do much harm to the runtime of the super-B-tree algorithm.

The formal description of the GCAS procedure will be given in the next paragraph. My recommendation is that the reader should only skim that subroutine description because the GCAS procedure is a very simple computer program whose fundamental characteristics were mostly described by the previous paragraph.

<u>Subroutine 3.4.11.</u> Let v and J denote the current node and integer that were described in the previous two paragraphs, let U denote a user-supplied module which indicated how an SDS field should be modified when it gains or loses a leaf descendant, let w denote the amount of runtime that U needs to insert or delete <u>a single leaf</u> in an SDS field, and let v* denote an anticipated node in the ANTICIP(v) set. Let us further assume that v* has a partially constructed SDS field and that this field describes that subset of

v*'s (anticipated) descendants which satisfy the usual condition of

4) $\quad INF(v^*) \leq KEY(y) \leq TEMP(v^*)$.

Let us also recall that the previous parts of this section indicated that PTEMP(v*) would denote a pointer to that y-record whose Key value equals TEMP(v*). <u>The GCAS procedure will use a three step algorithm for extending the SDS fields of every node, v*, which is a member of ANTICIP(v) and which additionally has a partially constructed SDS field.</u> This three step procedure is described below:

1) The first step of GCAS will make PTEMP(v*) walk J position rightward through the super-B-tree's list of y-leaves. This step will make a subroutine-call to module U every time it encounters a y-leaf that satisfies the inequality of $KEY(y) \leq SUP(v)$. The cited subroutine will be instructed to insert these leaves into the SDS(v*) field.

2) The second step will update the value of TEMP(v*) so that it reflects the insertions that were made in the previous step.

3) The third step will check to see whether TEMP(v*) equals SUP(v*). If this condition is satisfied then a flag will be raised over the SDS field (to indicate that this field is now "fully constructed").

Observation 3.4.L Let us assume that super-B-tree T initially represents the set S of leaves. It is trivial to confirm that under these circumstances a subroutine-call to the GCAS procedure will transform this tree into a new data-structure which is also a "representation" of the same set, S, of leaves. The only difference between the new and old data-structures will obviously be that the new data-structure will have J new leaves inserted into the partially constructed SDS fields of the members of the ANTICIP(v) set. It is also easy to see that such a subroutine-call to GCAS will require no more than 6jw time. A more formal theorem and proof about the GCAS procedure will not be included in this thesis because the procedure is completely trivial.

The mainline of the super-B-tree procedure will be called $Alg(\alpha, \beta, K, J)$. This procedure will have a very similar purpose to $Alg(\alpha, K)$, and its goal will thus be to insert or delete any leaf in a super-B-tree which the user commands.

The four parameters of the $Alg(\alpha, \beta, K, J)$ procedure will play a major role in the rest of this chapter. This is because the $Alg(\alpha, \beta, K, J)$ procedure will attain its needed $\Theta(w \log N)$ worst-case runtime only when great care is taken in assuring that it is assigned the proper set of initial parameter values. Much of the discussion in this section will center around these parameters. The next several paragraphs will offer a brief summary of the meaning and significance of the α, β, K and J parameters.

The parameter of J will represent the argument which the $Alg(\alpha, \beta, K, J)$ procedure shall pass on to its GCAS subroutine. A detailed description of the GCAS subroutine was given in the last several paragraphs of this section. That discussion indicated that the GCAS subroutine would insert J new leaves into each of the partially constructed SDS fields that it processes. Intuitively, J will thus represent the speed at which partially constructed SDS fields are converted into fully constructed fields. A more detailed description of the significance of J will be given later in this chapter during the definition of the $Alg(\alpha, \beta, K, J)$ mainline.

The next two parameters of this mainline will be α and K. For the most part, the $Alg(\alpha, \beta, K, J)$ procedure will employ these parameters in the same manner as did the $Alg(\alpha, K)$ mainline. Both of these parameters will thus be passed on to the $DETERMINE(\alpha, K)$ subroutine. The latter procedure was described earlier in part 3.2.K of this chapter. The previous discussion indicated that $DETERMINE(\alpha, K)$ was designed for the occasions when the value of $p(v)$ has moved outside the $(\alpha, 1-\alpha)$ range. That discussion indicated that the purpose of this subroutine was to use the α and K parameters to decide which one of the R_1, R_2, R_1' or R_2' should be applied to force $p(v)$ back into the $(\alpha, 1-\alpha)$ range.

The last parameter of the $Alg(\alpha, \beta, K, J)$ procedure will be β. This parameter will always be assumed to be less than α, and it will perform its assigned function only when the α and K parameters satisfy certain

special conditions that will be discussed later in part 3.6.A of this chapter. Under these circumstances, the parameter of β will place a special "bound" on our super-B-tree which will insure that all its current nodes of v will have a p(v) value that belong to the $[\beta, 1-\beta]$ interval.

A full description of the role of the α, β, K and J parameters will be given during the formal definition of the $Alg(\alpha, \beta, K, J)$ procedure that takes place in part 3.4.J of this chapter. It is possible to give a slightly more intuitive explanation of the role that these parameters will play in the $Alg(\alpha, \beta, K, J)$ mainline if this procedure is compared with section 3.2's $Alg(\alpha, K)$ mainline.

The main distinction between these algorithms can be understood if it is recalled that the $Alg(\alpha, K)$ mainline will automatically apply that rotation which the DETERMINE(α, K) subroutine has recommended for node v as soon as p(v) has moved outside the specified $(\alpha, 1-\alpha)$ interval. The $Alg(\alpha, \beta, K, J)$ procedure will differ from $Alg(\alpha, K)$ by being much more hesitant in its application of this rotation. This mainline will be formally defined in part 3.4.J of this section. That definition will insure that this procedure will only apply that rotation of R which the DETERMINE(α, K) subroutine has recommended for node v when the SDS fields of the ANTICIP(v, R) set have reached fully constructed states. Section 3.6 will explain how this requirement along with certain other requirements will insure that the $Alg(\alpha, \beta, K, J)$ parameter will manipulate its SDS fields in the

desired $\Theta(w \log N)$ worst-case runtime. That discussion will also explain

i) how the parameter of J will control the time at which rotation R is performed by determining the speed at which the SDS fields of the ANTICIP(v, R) set are built

ii) how the parameter of β together with the appropriately chosen accompanying α, K, and J parameter will insure that if node v has more than a minimal number of descendants then the $Alg(\alpha, \beta, K, J)$ mainline will perform that rotation which is recommended by the DETERMINE(α, K) subroutine sometime during the interim period when p(v) has moved outside the $(\alpha, 1-\alpha)$ interval but still lies inside the $[\beta, 1-\beta]$ interval.

During this chapter's detailed discussion of the $Alg(\alpha, \beta, K, J)$ procedure, the reader should bear in mind that the objective of this algorithm will be to insure that it will have an $\Theta(w \log N)$ runtime even in worst-case circumstances. There will be several special cases that will have to be considered in this chapter to ascertain that this worst-case runtime will prevail. For example, several special requirements will be placed on the α, β, K and J parameters to solve the unusual rotation problems that arise when the $Alg(\alpha, \beta, K, J)$ procedure attempts to apply near-simultaneous rotations to node v and one or more of its sons or grandsons.

Finally, the reader should be once again reminded that he should pay

special attention to the role of the α, β, K and J parameters during this chapter's analysis of the runtime of the Alg(α, β, K, J) mainline. This is because this algorithm will attain its highly desired Θ(w log N) runtime only when it is given a very carefully selected set of parameter values. The super-B-tree algorithm will thus later be defined to be that very special version of Alg(α, β, K, J) which results when this procedure is given this carefully selected set of parameters.

This chapter's remaining discussion of the Alg(α, β, K, J) procedure will be divided into three parts which will include:

i) a formal description of the Alg(α, β, K, J) mainline (in part 3.4.J of this section);

ii) a proof about the correctness of this procedure (in section 3.5);

iii) and a proof about its runtime (in section 3.6).

The reader should be forewarned that he will probably not fully understand the intuitive reason for the complex definition of the Alg(α, β, K, J) procedure until he reaches section 3.6. This is because section 3.5 and the other earlier parts of this chapter are only related to proving that this procedure can correctly perform those insertion and deletion operations which the user commands. The full importance of the Alg(α, β, K, J) procedure will not become apparent until section 3.6 shows how an extremely carefully selected set of α, β, K and J parameter values will produce an algorithm that is not only correct but

also has the needed Θ(w log N) worst-case runtime.

The formal description of Alg(α, β, K, J) is given below:

Algorithm 3.4.J. Let Q(K) denote the same quadratic which was originally described in equation 8 of section 3.2 and which is reproduced below

5) $$Q(K) = 1 - \frac{\sqrt{K^2 - K}}{K} \quad .$$

Let us assume that α and K satisfy the same two conditions which the Alg(α, K) procedure previously required them to satisfy. These two conditions are listed below:

6) $$K \geq 2$$

7) $$0 \leq \alpha \leq Q(K) \quad .$$

Let us further assume that J is a positive integer and that β is a real number which satisfies the condition of

8) $$0 < \beta < \alpha < 1/3 \quad .$$

The preceeding four quantities of α, β, K, and J should be regarded as four fixed numbers which are necessary so that the Alg(α, β, K, J) procedure can become fully defined. When defined, this procedure will require five further arguments so that it can execute a leaf insertion or leaf deletion command. These additional arguments will be

i) an initial super-B-tree that shall be denoted as T and which will normally be presumed to satisfy the condition that each current interior node, v, of T will have a $p(v)$ value such that:

9) $$\beta \leq p(v) \leq 1 - \beta$$

ii) an "SDS update module" of U which will indicate how the user wishes the SDS field of any node v to be modified when that node either gains or loses a single descendant y-leaf

iii) a parameter of w which indicates the amount of runtime that module U will need to perform the relevant SDS update operation on an arbitrary interior node

iv) a present (or future) leaf in our super-B-tree that shall be denoted as y

v) and a command which shall be denoted as c and which shall either state that leaf y should be inserted into our super-B-tree or that it should be deleted from this super-B-tree.

The general purpose of the $Alg(\alpha, \beta, K, J)$ procedure will be to perform those modifications in super-B-tree T which are suggested by three arguments of c, y, and U and which will additionally assure that all the $p(v)$ values of the interior nodes of this tree will continue to satisfy equation 9 at the time when this procedure is done executing. The specific algorithm that is used by $Alg(\alpha, \beta, K, J)$ will consist of the following 3 parts:

STEP 1: The first step of $Alg(\alpha, \beta, K, J)$ will be designed to physically insert (or delete) that leaf which is specified by the user's insertion (or deletion) command of c. The more or less obvious tree manipulating procedure will be used to perform the specified insertion (or deletion) operation. This procedure will begin with a standard top down tree search where the algorithm attempts to locate that part of super-B-tree T where the Key associated with the designated leaf of y should be stored. The second part of this step will physically modify the information at this location by performing that leaf insertion (or leaf deletion) operation which is suggested by the user's initial insertion (or deletion) command of c.

STEP 2: The second part of the $Alg(\alpha, \beta, K, J)$ procedure will be designed to update the SDS fields of super-B-tree T so that these fields will reflect that insertion (or deletion) operation which step 1 performed. The specific procedure that will be used by this step is trivial, but unfortunately it will require a lengthy discussion to explain how this procedure will handle the several different required cases. The detailed description of this basically trivial procedure will take place in part 3.5.D of this chapter. The procedure that is described in that part will be henceforth called the USDS subroutine. This subroutine will be designed to insure that all the "current" and "anticipated" SDS fields will reflect that insertion (or deletion)

operation which step 1 has performed.

STEP 3: Let v denote a current interior node that lies in super-B-tree T and is an ancestor of that leaf y (which step 1 either inserted or deleted). Note that the step 1 procedure is capable of causing the $p(v)$ ratios of such ancestor nodes to attain new values that lie outside the required $[\beta, 1-\beta]$ interval. The goal of this step will be to sufficiently "rebalance" such ancestor nodes to insure that their $p(v)$ values will be returned to the required $[\beta, 1-\beta]$ interval at the time when this step is done executing.

The specific procedure that is used by step 3 will be carefully designed to be efficient and to try to additionally confine the value of $p(v)$ to the more restrictive $[\alpha, 1-\alpha]$ interval (when possible). This step will be the only aspect of the $Alg(\alpha, \beta, K, J)$ procedure that is genuinely subtle. The procedure in this step will consist of a large DO-LOOP that is designed to process the ancestors of the initial leaf of y in a bottom up order. For each such current interior node ancestor of v, this DO-LOOP will execute the following three substeps:

SUBSTEP 3a: The first substep will be designed to enlarge the partially constructed SDS fields of those anticipated nodes which belong to the ANTICIP(v) set of ancestor v. This substep will rely upon a subroutine-call to the previously discussed GCAS

procedure to do this work. The GCAS procedure will be given the three arguments of v, J and U. The response of GCAS subroutine will be to instruct module U to insert J new leaves into the "partially constructed" SDS fields which are associated with the ANTICIP(v) set. The purpose of such an action will be to continue the process where our algorithm "gradually" constructs those SDS fields which it "anticipates" it will need in the near future. A more formal description of the precise mechanic of this substep's GCAS subroutine can be found in the earlier part of this chapter which was given the label of 3.4.H.

SUBSTEP 3b: The second substep will only be executed if ancestor v has a $p(v)$ value that lies outside the $(\alpha, 1-\alpha)$ interval but inside the $[\beta, 1-\beta]$ interval. Under these circumstances, this step will contemplate rebalancing v with a R_1, R_2, R_1' or R_2' rotation for the purposes of giving this node a $p(v)$ value that lies inside the $(\alpha, 1-\alpha)$ interval. The relevant rotation will be performed by this step if the SDS fields that it requires have been previously raised to a fully constructed state. A more detailed description of this substep can be given if we divide it into two suboperations. These two suboperations are described below:

i) The first suboperation will consist of a subroutine-call to the DETERMINE(α, K) procedure (which was previously described in part 3.2.K). This subroutine-call will instruct the DETERMINE(α, K) procedure to decide which one of the R_1, R_2, R_1' or R_2' rotations should be used to rebalance node v so that it can regain a p(v) value that lies inside the (α, 1 - α) interval. In the rest of our discussion, the symbol of R will denote that one of the R_1, R_2, R_1' or R_2' rotations that is recommended by DETERMINE(α, K).

ii) The second suboperation of substep 3b will be designed to apply rotation R to node v IF all the members of the ANTICIP(v, R) set have "fully constructed" SDS fields. The specific procedure used by this suboperation will have two parts. The first part will perform a straightforward check on the members of the ANTICIP(v, R) set to determine whether all its SDS fields are "fully constructed". The second part of the present procedure will only be activated if the preceding check indicated that all the

required SDS fields are "fully constructed". In that case, the second part will instruct a subroutine, called ROTATE, to apply rotation R to node v. A formal description of the ROTATE procedure will be given later in part 3.6.F of this chapter. This subroutine will be shown to be a fairly simple procedure that will use the "fully constructed" SDS fields of the ANTICIP(v, R) set to help it perform rotation R in $\Theta(w)$ worst-case runtime. It should be restated that the preceeding subroutine-call to ROTATE will only be made if all the SDS fields of ANTICIP(v, R) are fully constructed. If these conditions do not prevail then our algorithm will postpone rotation R to a later date in time when these conditions are satisfied.

SUBSTEP 3c: This substep will be the last part of the procedure which step 3 uses to rebalance ancestor nodes similar to v. This substep will ONLY be activated in that EMERGENCY case where p(v) has attained a value that lies outside the required (β, 1 - β) interval. In that case, this substep will use a subroutine called PENALTY-P to force the value of p(v) back

into the $[1/3, 2/3]$ interval (which will always be a proper subset of the $[\alpha, 1-\alpha]$ and $[\beta, 1-\beta]$ interval). A formal description of the PENALTY-P subroutine will be given in part 3.5.G of this chapter. That discussion will explain how subroutine-calls to PENALTY-P will be truely "penalizing" because its invocations will be frequently expensive. Much of the discussion in this section will center around the question of how the $Alg(\alpha, \beta, K, J)$ mainline can minimize the damage caused by these potentially expensive invocations of PENALTY-P. The main theorem of this chapter will state that the PROPER CHOICE of values for the α, β, K and J parameters will sufficiently minimize the number of these subroutine-calls to insure an $\Theta(w \log N)$ worst-case runtime for the $Alg(\alpha, \beta, K, J)$ procedure. This choice of α, β, K and J parameters will, in other words, enable us to define the desired super-B-tree algorithm.

The rest of this chapter will be devoted to more fully describing the subroutines that are called by the $Alg(\alpha, \beta, K, J)$ procedure as well as proving the correctness of this procedure and showing how the proper set of α, β, K, and J parameters will enable it to attain its proclaimed $\Theta(w \log N)$ worst-case runtime. The first two topics will be discussed in section 3.5, and the third topic will be discussed in section 3.6.

3.5 The Proof of the Correctness of $Alg(\alpha, \beta, K, J)$

This section will have two purposes. The first will be to prove that the $Alg(\alpha, \beta, K, J)$ procedure can correctly execute the user's leaf insertion and deletion commands. The second will be to give a more detailed description to the USDS, ROTATE and PENALTY-P subroutines that are called by this algorithm. All the concepts in this section are extremely important because they will be used during the next section's analysis of the runtime of the $Alg(\alpha, \beta, K, J)$ procedure. The discussion will now begin with several preliminary definitions:

Definition 3.5.A. Let us recall that Definition 3.4.E indicatated that a "partially constructed" SDS field of node v was defined to be any SDS structure which describes a proper subset of v's set of leaf descendants. Note that the preceeding definition implied that such "partially constructed" SDS structure can contain as few as one single leaf member. If this SDS field does contain only one leaf then it will be said to be a SDS field that is in its "starting state."

Comment: It is easy to see that Definitons 3.5.A and 3.4.E jointly imply that v's SDS field will be in its "starting state" if and only if that SDS field describes the single leaf whose Key value equals INF(v).

Definition 3.5.B. Let z denote a "current" interior node in a super-B-tree. Let f denote that node which is either the father or grandfather of z.

Let v^* denote an anticipated node that belongs to the ANTICIP(f) set. Node v^* will be said to be "dependent" on node z if the application of one of the R_1, R_2, R_1' or R_2' rotations to node f will simultaneously have v^* as a renewal node and z as a removal node.

Example 3.5.C. An examination of the previous Diagrams 3.2.D through 3.2.G indicates that

i) Nodes v_A, v_B, and v_C will be dependent on v_D;

ii) The latter two nodes of v_B and v_C will also be dependent on v_B;

iii) Nodes v_A', v_B' and v_C' are dependent on v_D';

iv) The latter two nodes of v_B' and v_C' will also be dependent on v_B'.

The preceding definitions will be important because they will help characterize the nature of the USDS, ROTATE and PENALTY-P subroutines. These subroutines will be discussed in the next three lemmas. The purpose of these lemmas will be to show that it is possible to design USDS, ROTATE and PENALTY-P subroutines that operate with moderate efficiency and perform the tasks which are required respectively by steps 2, 3b and 3c of the $Alg(\alpha, \beta, K, J)$ mainline. The proofs of the three cited lemmas will confirm their propositions by first outlining how the needed subroutine should

be designed and by secondly showing how these subroutines perform their needed tasks in the claimed runtime. It should be stated that the proofs of the next three lemmas will be quite trivial, and some readers may consequently prefer to skim or omit them. If the reader does decide to omit these proofs then he will more rapidly approach the more subtle and important theorems of the next section.

LEMMA 3.5.D. Let us recall that the purpose of step 2 of the $Alg(\alpha, \beta, K, J)$ procedure was to modify the SDS fields (of the current and anticipated nodes) of our super-B-tree so that these fields would reflect that leaf insertions or leaf deletion operation which step 1 executed. It is possible to design a procedure (called USDS) which will perform these operations in $\Theta(w \log N)$ runtime for every super-B-tree that has an $\Theta(\log N)$ height.

The proof of Lemma 3.5.D will be the most trivial of the four proofs that are given in this section. Unfortunately, that proof will also be very long because it will require us to formally define the (USDS) subroutine whose existence is claimed. Frankly, the reader's time may be better spent if he omits examining the details of this proof. This is because the USDS subroutine will manipulate partially and fully constructed SDS fields of anticipated and current interior nodes in the obvious manner which is suggested by the definitions of these concepts. The only substantial advantage that the reader may obtain from examining the proof of Lemma 3.5.D is that it may help

further familiarize him with the terminology of this chapter.

Proof of Lemma 3.5.D: The proof will be divided into two parts which separately verify that it is possible to design the two parts of the USDS subroutine which are needed to properly update the SDS fields of the current and anticipated nodes in the claimed $\Theta(w \log N)$ runtime. In the discussion which follows, the readers should bear in mind that it is the user's responsibility to supply the $Alg(\alpha, \beta, K, J)$ mainline with a procedure of U that indicates how the SDS fields should be revised when a node gains or loses a leaf descendant. The mere goal of the USDS subroutine will be to decide when and where the module of U should be applied.

The discussion of the USDS subroutine will begin with a proof that it is possible to design a procedure whose purpose is to perform USDS's first task of updating the SDS fields of the current interior nodes in the proclaimed $\Theta(w \log N)$ runtime. All parts of this aspect of the USDS subroutine are trivial. A two-part procedure will be used to perform this trivial task. The first part will undertake a standard bottom up tree walk in search of those current interior nodes v which are an ancestor of that leaf y which was specified in the user's initial insertion or deletion command. The second part of the current node update procedure will make subroutine-calls to the user's module of U for every node of v that the first step located. The purpose of these subroutine-calls will be to instruct U to appropriately

update the designated SDS fields. It is absolutely trivial to verify that the $\Theta(\log N)$ height of tree T implies that such a two-step procedure will succeed in correctly updating all the SDS fields of T's current interior nodes in the claimed $\Theta(w \log N)$ worst-case runtime.

The second part of the USDS procedure will be designed to similarly update the SDS fields of T's anticipated interior nodes. A four-step procedure will be used to perform this task. The first step will once again be a bottom up tree walk that is designed to find all those current interior nodes v which are an ancestor of that y-leaf which was specified in the user's initial insertion or deletion command. The second step will scan the ANTICIP(v) sets of the preceeding current nodes for the purposes of producing the list of anticipated nodes, v^*, that belong to these sets. The third step will be designed to construct the specific list of anticipated SDS fields that require updating. It will do this by comparing the $INF(v^*)$, $SUP(v^*)$ and $TEMP(v^*)$ components of step 2's anticipated nodes with the Key of that y-leaf which was given in the user's initial insertion or deletion command. The specific procedure in this third step will produce the list of SDS fields that will satisfy one of the following two conditions:

i) $SDS(v^*)$ field is "fully constructed" and $INF(v^*)$ $INF(v^*) \leq KEY(y) \leq SUP(v^*)$

ii) or $SDS(v^*)$ is "partially constructed" and $INF(v^*) \leq KEY(y) \leq TEMP(v^*)$.

Finally, the fourth step will take step 3's SDS fields and instruct module U to revise these SDS fields in accordance with the user's initial insertion or deletion command.

It must be emphasized that it is trivial to confirm the above procedure will properly update the anticipated SDS fields. This is because a straightforward analysis will reveal that

A) If SDS(v*) requires updating then it must pass the tests of steps 2 and 3.

B) If SDS(v*) passes the test of step 3 then it must require updating.

The preceeding two observations can easily be used to verify that all anticipated SDS fields are properly updated. Also, the $\Theta(\log N)$ height mentioned in the hypothesis of Lemma 3.5.D implies that the USDS subroutine can perform its anticipated SDS update task in the same $\Theta(w \log N)$ runtime that was previously used by the current interior node task. The last sentence completes this admittedly trivial proof because that sentence indicated that it is possible to design a USDS subroutine which correctly updates the current and anticipated SDS field in the claimed $\Theta(w \log N)$ worst-case runtime.

Q.E.D.

Observation 3.5.E. Let us recall that part 3.4.G of this chapter gave the definition of a data-structure which is a super-B-tree "representation" of a specified set of leaves. Let S_0 denote an initial set of leaves, T denote

a super-B-tree whose data-structure is a "representation" of S_0, c denote a command which either indicates that record y should be inserted or deleted, and S_1 denote the modification of the S_0 set which results when command c is executed. Note that the procedure which was used by step 1 of $Alg(\alpha, \beta, K, J)$ was adequately discussed during part 3.4.J's description of this mainline and that the step 2 procedure was described by the previous lemma. It is easy to verify that the net effect of these two steps will be to sufficiently modify the data-structure of tree T so that it is transformed from an initial representation of the S_0 set of leaves to a representation of the S_1 set of leaves. A formal proof of this observation will not be included in this thesis because it is absolutely trivial to verify. The significance of this trivial observation will become apparent at the end of this section.

LEMMA 3.5.F. It is possible to disign a ROTATE subroutine which can always perform in $\Theta(w)$ runtime that rotation which is requrested by operation ii) of substep 3b of the $Alg(\alpha, \beta, K, J)$ mainline. A more specific description of this task can be given if S_1 denotes the same set of leaves which were mentioned in 3.5.E, T denotes a super-B-tree data-structure which represents this set of leaves, v denotes a current interior node of T, and R denotes one of the R_1, R_2, R_1' or R_2' rotations (which were preciously illustrated in Diagrams 3.2.D through 3.2.G). Note that the

ROTATE subroutine will be invoked by substep 3b (of the Alg(α, β, K, j) procedure) in the context of a request to apply rotation R to node v of super-B-tree T. The specific claim of this lemma is that it is possible to design a ROTATE subroutine that has an $\Theta(w)$ runtime and whose invocation will produce a new data-structure for tree T that represents the same S_j set of leaves as the old structure and which will differ from the old structure in the following three respects.

1) The "current" interior nodes of tree T will be modified by the ROTATE subroutine in the obvious manner that is suggested by the arguments of v and R. This fact basically means that ROTATE subroutine will alter T's current interior node structure by replacing R's removal nodes with its renewal nodes (as was previously illustrated in Diagrams 3.2.D through 3.2.G).

2) The SDS fields of the current nodes in tree T will be modified by the ROTATE subroutine to reflect the changes that this subroutine has made in the interior node structure of this tree. (This condition basically means that the ROTATE subroutine will construct those new SDS fields that are associated with the renewal nodes that are generated by rotation R).

3) Thirdly, the ROTATE subroutine will make the SDS fields

of the anticipated nodes also reflect the fact that rotation R has been performed. The specific nature of this modification can be best explained if v* denotes an anticipated node in our super-B-tree. The ROTATE subroutine will adjust the SDS fields of three types of anticipated nodes to reflect the fact that rotation R has been performed. The anticipated nodes that will need adjustment will be:

i) That set of v* nodes which belong to the ANTICIP(v) set;

ii) The similar anticipated nodes that belong to the anticipated sets of R's renewal nodes.

iii) Those special v* anticipated nodes which are "dependent" on v. (Definition 3.5.B implies that these "dependent" nodes will belong to the anticipated sets of v's father and grandfather.)

The ROTATE subroutine will cause the SDS fields of all three of the preceeding classes of anticipated nodes to be put into their starting states (so as to reflect the fact that rotation R has been performed).

The basic claim of this lemma can be summarized as stating that it is possible to design a ROTATE subroutine which performs the preceeding three tasks

in $\Theta(w)$ runtime on each occasion when this subroutine is called by the $Alg(\alpha, \beta, K, j)$ mainline.

Proof: It is easiest to verify the above lemma if we separately examine the three characteristics that this lemma attributed to the ROTATE subroutine and show that it is possible to design a ROTATE subroutine which performs these three functions in the claimed $\Theta(w)$ worst-case runtime. The details of this three part analysis are given below:

1) It is trivial to design a procedure that performs the ROTATE subroutine's first task of altering the node structure of tree T in the manner suggested by rotation R. This is because such node replacement operations merely require the modification of a few simple pointers. Such pointer changing operations can obviously be performed within the required runtime.

2) A slightly more complicated argument is required to show that the ROTATE subroutine can perform its second task of constructing the SDS fields of the newly produced renewal nodes in the required $\Theta(w)$ runtime. Normally, such operations that produce SDS fields would require much more runtime. The ROTATE subroutine's special ability to efficiently produce these SDS fields is due to the fact that the $Alg(\alpha, \beta, K, j)$ mainline will ask the ROTATE subroutine to apply rotation R to node

v only after the mainline has confirmed that the corresponding SDS fields of the ANTICIP(v, R) set have reached fully constructed states. Clearly, these circumstances imply that the renewal nodes of rotation R can be assigned their new SDS fields by simply adjusting the pointers in these nodes so that they will contain the addresses of the corresponding SDS fields which were previously associated with the ANTICIP(v, R) set. The significant aspect about these pointer changing operations is that they consume $\Theta(1)$ worst-case runtime. The ROTATE subroutine can thus be designed to perform its second task in the required runtime.

3) It is trivial to design a procedure that enables the ROTATE subroutine to perform its third task in the required $\Theta(w)$ runtime. This is because the third task merely consisted of assigning starting state SDS field to 23 (or less) anticipated nodes. Each starting state SDS field will require no more than w time. All 23 fields can thus be produced in the required $\Theta(w)$ time.

One last remark must be made with regards to the present proof of the ROTATE lemma. Note that in addition to mentioning the above three tasks, this lemma also stated that the net effect of the ROTATE subroutine would be to alter tree T by giving it a new representation for the same S_i set of

leaves which it represented before the ROTATE subroutine was invoked. It is very easy to verify this last assertion. This is because the three tasks of the ROTATE subroutine can easily be shown to produce precisely such a modification in tree T. The proof has thus been completed since the ROTATE subroutine has been shown to perform the desired functions in the required $\Theta(w)$ worst-case runtime.

Q. E. D.

LEMMA 3.5.G. Let v denote a current interior node, N_v denote its number of descendants, and let us recall that the purpose of step 3c of the $Alg(\alpha, \beta, K, J)$ mainline was to reorganize the subtree which has v as a root so that every node d in this subtree is given a $p(d)$ value that belongs to the $[1/3, 2/3]$ interval. The basic claim of this lemma is that it is possible to design a subroutine (called PENALTY-P) which performs this super-B-tree task in $\Theta(w N_v \log N_v)$ runtime. A more specific description can be given to the function of the PENALTY-P subroutine if S_1 denotes the same set of leaves as it did in 3.5.E, T denotes a super-B-tree whose data-structure is a representation of this set of leaves, and v denotes a current node in tree T. The specific claim of this lemma is that it is possible to design a PENALTY-P subroutine whose invocation with the arguments of v and T will modify tree T so that it is assigned a new data-structure which represents the same S_1 set of leaves (as the old structure) and which will differ from the old representation in the following respects

1) That portion of tree T that is a descendant of node v will be entirely reorganized by the PENALTY-P subroutine to insure that if node d either equals v or is one of v's descendants then $p(d)$ will necessarily have a value that belongs to the $[1/3, 2/3]$ interval.

2) The SDS fields of the current nodes of tree T will be modified by the PENALTY-P subroutine so as to reflect those changes in the current interior nodes which condition 1) discussed. (This condition basically means that the PENALTY-P subroutine will produce a new SDS(d) field for every current node d which is a descendant of v.)

3) Thirdly, the PENALTY-P subroutine will make the SDS fields of the anticipated nodes of our super-B-tree reflect the other changes that were made in this tree. The specific nature of this modification can be best explained if v* denotes an anticipated node. The PENALTY-P subroutine will be required to adjust the SDS fields of two types of anticipated nodes in response to the other changes it made in tree T. These two types of anticipated nodes will be

   i) That class of v* anticipated nodes that belong to an ANTICIP(d) set of a node, d, which is

either equal to v or is a descendant of v.

ii) Those special v^* anticipated nodes which are dependent on v.

The PENALTY-P subroutine will cause the SDS fields of the preceeding two classes of anticipated nodes to be put into their starting states (in order to complete the changes that are required by the modified tree structure which the PENALTY-P subroutine has created).

The basic claim of this lemma can be summarized as stating that it is possible to design a PENALTY-P subroutine which performs the above three tasks in $\Theta(w N_v \log N_v)$ runtime.

Proof: All aspects of the PENALTY-P subroutine are trivial. Consequently, this proof will contain a very brief discussion of this subroutine.

Let us begin by examining the first task which the PENALTY-P subroutine is required to perform. That task consisted of a reorganizing the subtree which has v as a root so that every node, d, in this subtree will have a $p(d)$ value that belongs to the $[1/3, 2/3]$ interval. It is very easy to design a procedure that performs this task. This is because the v-rooted subtree can be made to satisfy the cited condition if it is reorganized so that every one of its nodes of d are given left and right sons whose leaf-weights

differ by no more than an integer of 1. Such a new tree can be produced in $\Theta(N_v \log N_v)$ runtime by any one of several standard sort algorithms. Thus, any one of several standard sort algorithm will enable the PENALTY-P subroutine to perform its first task in the required runtime.

The second and third parts of the PENALTY-P subroutine are even more trivial than the preceeding procedure. These aspects of the PENALTY-P subroutine will be required to make the SDS fields reflect those modifications which were made by the first part of this procedure. It is absolutely trivial to design procedures that perform these tasks in their required $\Theta(w N_v \log N_v)$ runtimes. This proof has thus verified that PENALTY-P can perform all three of its required tasks in this runtime.

One final remark is necessary to complete the proof of the PENALTY-P lemma. Note that this lemma stated that the net effect of a subroutine-call to PENALTY would be to convert one representation of the S_1 set of leaves into a second equally eligible representation of this set of leaves. It is trivial to verify this last statement because obviously the three components of the PENALTY-P subroutine will perform precisely such a task. Q. E. D.

Comment: It must once again be emphasized that the PENALTY-P subroutine is an inefficient procedure and that the preceeding proof was completely trivial because this subroutine performed no task with noteworthy efficiency. Most of the rest of this section will study the question of how to avoid the inefficiency

PENALTY-P subroutine. Once again, the reader should be reminded that the main goal of the forthcoming discussion will be to find that set of α, β, K and J parameters which enable the Alg(α, β, K, J) procedure to attain its Θ(w log N) runtime by minimizing its subroutine calls to PENALTY-P.

The final topic of this section will be Theorem 3.5.H. That proposition will complete the first half of the discussion of the Alg(α, β, K, J) procedure by showing that this algorithm correctly execute the user's leaf insertion and leaf deletion commands. As the reader examines this theorem, he should bear in mind that this proposition does not make any mention about the runtime of the Alg(α, β, K, J) procedure and that this second topic will be discussed in section 3.6.

THEOREM 3.5.H. Let S_0 denote an initial set of leaves, c denote an insertion or deletion command, S_1 denote the modification of the S_0 set that results from command c, and T denote a super-B-tree that is initially a representation of the S_0 set of leaves. Under these circumstances, the Alg(α, β, K, J) procedure will "correctly" modify tree T when it is given this command of c. More specifically, the net effect of such an invocation of the Alg(α, β, K, J) procedure will be to revise tree T so that it becomes a representation of the S_1 set of leaves.

Proof: The proof of this theorem is a completely trivial consequence of the

previous assertions. The proof will begin with the observation that part 3.5.E of this section indicated that the net effect of Steps 1 and 2 of the Alg(α, β, K, J) procedure would be to sufficiently modify tree T so that it becomes a representation of the S_1 set of leaves. The remaining proof of the present proposition will be based on the observation that the only parts of the Alg(α, β, K, J) which will subsequently further modify the data-structure of tree T are step 3's the GCAS, ROTATE and PENALTY-P subroutines. Note that Observation 3.4.I, Lemma 3.5.F, and Lemma 3.5.G collectively indicated that the net effect of a subroutine-call to any one of these three procedures would be to convert one data-structure which represents the S_1 set of leaves into another data-structure which represents the same set of leaves. The clear implication of these facts is that tree T must continue to be a representation of the S_1 set of leaves at the time when the Alg(α, β, K, J) procedure is done executing.

Q. E. D.

Comment. Some readers may have been surprised by the previous proof because it indicated that the only result of a subroutine-call to the GCAS, ROTATE, or PENALTY-P procedures would be to perform the relatively neutral operation of converting one correct representation of the S_1 set of leaves into another equally correct representation of this same set of leaves. The importance and purpose of these three subroutines will become more apparent in section 3.6. That section will show how the Alg(α, β, K, J) main-

line and its subroutines were carefully designed to assure that an $\Theta(w \log N)$ worst-case runtime will result (when the α, β, K, and J parameters are given the proper set of initial values). By the end of section 3.6, it will become clear why the Alg(α, β, K, J) procedure was defined in the special manner that it was. This is because all the complicated and subtle aspects of this procedure are related to the goal of achieving an $\Theta(w \log N)$ worst-case runtime.

3.6 An Analysis of the Runtime of Alg(α, β, K, J)

The purpose of this section will be to prove that the proper set of α, β, K and J parameters will cause the Alg(α, β, K, J) procedure to operate in $\Theta(w \log N)$ worst-case time. The discussion in this section will be divided into two parts. The first half of this section will assume (without proof) that the claim which is stated below in 3.6.A is valid. Given that assumption, this part of this section will derive the chapter's main theorem which will state that the proper α, β, K and J parameters will produce the desired $\Theta(w \log N)$ worst-case runtime. The second half of this section will complete the discussion of this runtime by verifying claim 3.6.A. The statement of this claim is given below.

Claim 3.6.A. There will exist numbers of Δ, α, β, and K such that the first three of these numbers will satisfy the condition of

10) $$1/3 > \Delta > \alpha > \beta > 0$$

and such that these four numbers will satisfy the additional condition that if v is a node whose p(v) value lies inside either the $[\beta, \alpha]$ or $[1-\alpha, 1-\beta]$ intervals, if additionally all the sons and grandsons, s, of v will satisfy the condition that their p(s) values lie inside the $[\beta, 1-\beta]$ interval, and if the DETERMINE(α, K) subroutine recommends that rotation R be used to rebalance node v then this rotation will

i) sufficiently modify the value of $p(v)$ to insure that it lies inside the $(\Delta, 1-\Delta)$ interval;

ii) and this rotation will produce renewal nodes of s^* whose $p(s^*)$ values also lie in the same $(\Delta, 1-\Delta)$ range.

The proof of the preceeding claim is quite long, and it will therefore be postponed until the later part of this section. The immediate goal of this section will be to utilize the preceeding claim to show that the proper set of α, β, K and J parameters will cause the $Alg(\alpha, \beta, K, J)$ procedure to operate in $\Theta(w \log N)$ worst-case runtime. The discussion of this topic will begin with several preliminary lemmas. The first of these lemmas is given below:

LEMMA 3.6.B. Let Δ, α, β, and K denote four numbers which satisfy the preceeding claim 3.6.A, J denote any arbitrary nonnegative integer, and T denote a super-B-tree whose every current interior node of v has a $p(v)$ value that lies in the $[\beta, 1-\beta]$ interval. Let us assume that the user gives a command to the $Alg(\alpha, \beta, K, J)$ procedure that either indicates leaf y should be inserted into or deleted from this tree. Under these circumstances, the $Alg(\alpha, \beta, K, J)$ procedure will modify tree T in a manner that will assure that the $p(v)$ values of all current interior nodes will continue to lie in the $[\beta, 1-\beta]$ range at the time when this procedure is done executing.

Proof: Note that the hypothesis of this lemma indicated that all current interior nodes, v, will initially have $p(v)$ values that lie inside the $[\beta, 1-\beta]$ interval. It consequently follows that the proof of Lemma 3.6.B will only require examination of the subset of tree T's nodes whose $p(v)$ values are changed by the execution of the $Alg(\alpha, \beta, K, J)$ procedure. These $p(v)$ values must be shown to also belong to the $[\beta, 1-\beta]$ range at the time when the procedure is done executing.

The $Alg(\alpha, \beta, K, J)$ procedure was formally described in part 3.4.J of this chapter. This proof will begin with the observation that there are only three parts of this algorithm that will change the $p(v)$ values of those nodes that they process. These three components of the $Alg(\alpha, \beta, K, J)$ procedure are:

A: Step 1 which will change the $p(v)$ values of all those current interior nodes that are ancestors of y.

B: Step 3c whose PENALTY-P subroutine will produce new $p(v)$ values for all those nodes which belong to the new subtree that is created by this procedure.

C: Operation ii) of step 3b whose ROTATE subroutine will produce two or three altered nodes with new $p(v)$ values on each occasion when this subroutine is invoked.

The proof of the present lemma will rest on a separate examination of the three classes of interior nodes that are modified by the above three components of the $Alg(\alpha, \beta, K, J)$ procedure. The goal of this proof will be to show that all the $p(v)$ values of these three classes of nodes will continue to lie in the $[\beta, 1-\beta]$ interval at time when the $Alg(\alpha, \beta, K, J)$ procedure is done executing.

Let us begin by examining the first of these three

[illegible] current interior nodes. This class will consist of that set of interior nodes which are ancestors of y (and whose $p(v)$ values are consequently changed by the insertion or deletion of leaf y). In many instances, the procedure in step 1 will sufficiently alter the $p(v)$ values of these nodes to cause them to lie outside the $[\beta, 1-\beta]$ range (at the time when this step is done executing). The significant characteristic of such nodes is that the $Alg(\alpha, \beta, K, J)$ procedure has been carefully designed to assure that the PENALTY-P subroutine (in step 3c) will further process those ancestors of leaf y whose $p(v)$ values has moved outside the required $[\beta, 1-\beta]$ range. It is very easy to show that the combined implication of part 1 of Lemma 3.5.G and equation 10 is that the PENALTY-P subroutine will force these $p(v)$ values to return into the $[\beta, 1-\beta]$ interval. This fact in other words means that the PENALTY-P subroutine will repair the $p(v)$ values of those nodes that step 1 caused to exceed the $[\beta, 1-\beta]$ interval. Hence, step 1 cannot perform any action that causes the $Alg(\alpha, \beta, K, J)$ procedure to produce a

$p(v)$ value which lies outside the $[\beta, 1-\beta]$ range at the time when step 3 is done executing.

Let us now turn our attention to the second class of modified nodes. This class will consist of those nodes which are affected by the PENALTY-P subroutine. This class of nodes will consist of many of the nodes that belonged to the preceeding class plus other nodes as well as other nodes. The significant characteristic of the PENALTY-P subroutine is that part 1 of Lemma 3.5.G together with equation 10 implies that it will produce a subtree whose every node will have a $p(v)$ value that belongs to the $[\beta, 1-\beta]$ range. These nodes will thus also satisfy the required condition.

The proof will now be completed by examining the third class of altered interior nodes. This class will consist of those nodes that are affected by the ROTATE *subroutine. Claim 3.6.A clearly implies that these nodes will be* given $p(v)$ values that belong to the $[\beta, 1-\beta]$ interval. This fact completes the proof of Lemma 3.6.B since the last of the classes of modified nodes has also been shown to have the proper $p(v)$ values. Q.E.D.

COROLLARY 3.6.C. The tree T discussed in the preceeding lemma will necessarily have an $\Theta(\log N)$ height.

Proof: Note that the preceeding lemma indicated that the $p(v)$ values of all current interior nodes of tree T belonged to the $[\beta, 1-\beta]$ range and

that Nievergelt and Reingold have previously shown that the latter fact implies that T will have an $\Theta(\log N)$ height. The corollary is thus a trivial consequence of Nievergelt's and Reingold's previous observations. Q. E. D.

Note that the discussion in this chapter has so far omitted mentioning the significance of the parameter of J. This is because this parameter was unrelated to Theorem 3.5.H's proof about the correctness of the $Alg(\alpha, \beta, K, J)$ procedure, to Lemma 3.6.B's proof about the "bounded balance" of tree T, and to Corollary 3.6.C's statement about the $\Theta(\log N)$ height of tree T. The importance of the parameter of J and its interaction with the α, β, K and Δ parameters will be discussed in the next several lemmas and theorems. That discussion will show that the $Alg(\alpha, \beta, K, J)$ procedure will attain an $\Theta(w \log N)$ worst-case runtime when the Δ, α, β, and K parameters are chosen to satisfy the conditions of claim 3.6.A and when the parameter of J is chosen to satisfy the following two inequalities:

11) $$J \geq 3 + \frac{3}{\beta^2(\Delta - \alpha)}$$

12) $$J > 3 + \frac{3}{\alpha - \beta} \quad .$$

Let r denote a real number. In our forthcoming discussion, the symbol of $\lceil r \rceil$ will denote the least integer that is greater than or equal to r. This notation will be used in the next several lemmas.

LEMMA 3.6.D. Let r_1 and r_2 denote two real numbers such that $r_1 < r_2$, v denote a node in tree T, C denote a sequence of insertion and deletion commands which modify the set of leaves that descend from v, N_{1v} denote the number of leaves that belong to v's set of descendants before command sequence C is given, and N_{2v} denote the size of this set after command sequence C is executed. Let us also assume that no instruction during command sequence C causes the ROTATE or PENALTY-P subroutines to directly affect node v and that the net effect of command sequence C is either to cause the value of $p(v)$ to increase from an initial value that is less than r_1 to a final value that exceeds r_2 or to decrease from an initial value that is greater than r_2 to a final value that is less than r_1. Also, let us recall that the symbol of $|C|$ is used in this chapter to denote the number of commands in sequence C. Under these circumstances, the following two inequalities must hold:

13) $$|C| > \lceil (r_2 - r_1) N_{1v} \rceil$$

14) $$|C| > \lceil (r_2 - r_1) N_{2v} \rceil \quad .$$

Proof: It is absolutely trivial to show that the net change in the value of $p(v)$ must be less than both $\frac{|C|}{N_{1v}}$ and $\frac{|C|}{N_{2v}}$ (because of the restrictions that the hypothesis placed on sequence C). Also the hypothesis implies that this same change must be at least as large as $r_2 - r_1$. Furthermore, the definition of

$|C|$ implies that this symbol must be an integer. The combined implication of these three facts is that equations 13 and 14 must hold. Q. E. D.

The preceeding lemma will play a major role during the analysis of the runtime of the $Alg(\alpha, \beta, K, J)$ procedure. It is necessary for one more definition to be introduced before this analysis can begin.

Definition 3.6.E. Note that Lemmas 3.5.F and 3.5.G implied that the application of either the ROTATE or PENALTY-P subroutines to a son or grandson of current interior node v would cause the SDS fields of some of the members of the $ANTICIP(v)$ set to return back to their initial starting states (because these anticipated SDS fields were "dependent" on many of v's sons and grandsons). Such an application of the ROTATE or PENALTY-P subroutines to v's son or grandson will be said to have caused an "untimely disruption of node v" if either condition i) or its mirror image analog of condition ii) does hold

i) $p(v)$ has a value that belongs to the $[1-\alpha, 1-\beta]$ interval and the cited subroutine-call has caused the SDS fields of either the members of $ANTICIP(v, R_1)$ or $ANTICIP(v, R_2)$ sets to have returned to their starting states.

ii) $p(v)$ has a value that belongs to the $[\beta, \alpha]$ interval and the cited subroutine-call has caused the SDS fields of either the

members of the $ANTICIP(v, R_1')$ or $ANTICIP(v, R_2')$ sets to return to their starting state.

Comment: The preceeding definition was given the name of "untimely disruption" because that term very aptly described the intuitive idea behind this concept. Thus, the next several lemmas in this section will explain how the conditions of the preceeding definition will only be met when a major "disruption" destroys a large amount of the information in the anticipated SDS fields at the very unfortunate point in time when the $p(v)$ values indicate that this information is likely to be soon needed.

LEMMA 3.6.F. Let us once again assume that Δ, α, β, and K satisfy the condition of claim 3.6.A. Let us further assume that T denotes a super-B-tree whose every node of n has a $p(n)$ value that belongs to the $[\beta, 1-\beta]$ interval, v denotes a node in this tree, C once again denotes a sequence of insertion and deletion commands which are given to the $Alg(\alpha, \beta, K, J)$ procedure for the purposes of modifying the descendants of node v, and N_{2v} once again denotes the number of descendants which v will have at the end of this command sequence. Let us assume that $p(v)$ has values that either lie inside the $[\beta, \alpha]$ range during the entire execution of this command sequence or has values that lie inside the $[1-\alpha, 1-\beta]$ interval. Let us further assume that command sequence C satisfies the inequality of

15) $$|C| \leq \lceil \beta^2(\Delta - \alpha) N_{2v} \rceil \quad .$$

Under these circumstances, there can never be more than two untimely disruptions during sequence C.

Proof: It is sufficient to prove the lemma only for the case where $p(v)$ belongs to the $[1-\alpha, 1-\beta]$ interval (since the other case will follow from symmetry arguments). The proof for this case will be based on the following three observations:

1) Let v_D denote v's right son and v_E denote the left son of v_D (as was previously shown in rotation diagrams 3.2.D and 3.2.E). It is a trivial consequence of Definition 3.6.E that v_D and v_E are the only sons and grandsons of v that are capable of causing it to have an untimely disruption (given the previously mentioned assumption that $p(v)$ belongs to the $[1-\alpha, 1-\beta]$ interval).

2) Let s denote either the node of v_D or v_E. An examination of the $Alg(\alpha, \beta, K, J)$ mainline, Lemma 3.5.G, and claim 3.6.A reveals that a subroutine-call which asks PENALTY-P or ROTATE to process node s will be made only when $p(s)$ lies outside the $(\alpha, 1-\alpha)$ interval and that such a subroutine-call will cause $p(s)$ to move inside the $[\Delta, 1-\Delta]$ range.

3) Lemma 3.6.D and equation 15 easily imply that if either $p(v_D)$ or $p(v_E)$ lie inside the $[\Delta, 1-\Delta]$ interval (at the end of any single command in sequence C) then this value will remain inside the $[\alpha, 1-\alpha]$ range for the rest of this command sequence.

It is fairly easy to show that the preceeding three observations imply that node v will be caused to have no more than one untimely disruption from node v_D and no more than one untimely disruption from v_E. This node will thus experience no more than two untimely disruptions.

Q. E. D.

LEMMA 3.6.G. Let T once again denote a super-B-tree whose every node of n will have a $p(n)$ value in the $[\beta, 1-\beta]$ interval, v once again denote a specific node in this tree, C again denotes a sequence of insertion and deletion commands that is given to the $Alg(\alpha, \beta, K, J)$ procedure for the purposes of modifying v's set of descendant leaves, N_{2v} once again denote the number of descendants which v has at the end of sequence C, and let us again assume that Δ, α, β and K satisfy the conditions of claim 3.6.A. Let us further assume that J will satisfy the requirements of the previously mentioned equation 11 (which is reproduced below):

16) $$J \geq 3 + \frac{3}{\beta^2(\Delta - \alpha)}$$

and that sequence C is sufficiently long to satisfy the following condition:

17) $$|C| \geq \left\lceil \frac{3 N_{2v}}{J - 3} \right\rceil \quad .$$

Under these circumstances, it is impossible for the value of $p(v)$ to remain in either the $[\beta, \alpha]$ or $[1 - \alpha, 1 - \beta]$ intervals during the ENTIRE period in which sequence C is executed.

Proof: The lemma will be verified by means of contradiction. Our goal will thus be to show that a contradiction will arise if it is assumed that $p(v)$ will remain in either the $[\beta, \alpha]$ or $[1 - \alpha, 1 - \beta]$ interval during the entire execution of sequence C.

Let C^* denote the subsequence of the last $\left\lceil \frac{3 N_{2v}}{J - 3} \right\rceil$ commands that appear in sequence C. The desired contradiction will be obtained if we examine this sequence.

The significant characteristic of sequence C^* is that equation 16 implies that its length must satisfy the inequality of:

18) $$|C^*| \leq \left\lceil \beta^2 (\Delta - \alpha) N_{2v} \right\rceil \quad .$$

The above equation together with the hypothesis of Lemma 3.6.G will imply that Lemma 3.6.F is applicable to sequence C^*. That lemma will indicate that this sequence contains no more than two untimely disruptions. It thus follows that sequence C^* will contain a subsequence of $\left\lceil \frac{N_{2v}}{J - 3} \right\rceil$ consecutive

commands that contain no untimely disruptions. The significant characteristic of this disruption-free subsequence is that J elements will be added to v's anticipated SDS fields (by the GCAS subroutine) during the execution of each command in this subsequence. Consequently, it is easy to see that such a subsequence of $\left\lceil \frac{N_{2v}}{J - 3} \right\rceil$ commands will be sufficiently long to guarantee that if R denotes that rotation which is recommended by the DETERMINE(α, K) subroutine then the SDS fields of the ANTICIP(v, R) sets must become fully constructing during this subsequence. The latter fact is important because operation ii) of substep 3b of the Alg(α, β, K, J) mainline will make a subroutine-call to the ROTATE procedure whenever this condition is satisfied. Such a rotation will cause $p(v)$ to return into the $(\alpha, 1 - \alpha)$ interval (by claim 3.6.A). The desired contradiction has thus been obtained since command sequence C was unable to keep $p(v)$ inside the $[1 - \alpha, 1 - \beta]$ range.

Q.E.D.

THEOREM 3.6.H. Let us once again assume that the Δ, α, β and K parameters satisfy claim 3.6.A and that the parameter of J satisfies equation 11. Let T denote a super-li-tree, and let us assume that all insertion and deletion operations in life of tree T are performed by the Alg(α, β, K, J) procedure. Let v denote a current interior node in tree T, c denote a single insertion or deletion command, and N_v denote the number of leaves which descend from v after command c is performed. If command c

causes the PENALTY-P subroutine to be applied to the subtree whose root is v then the following equation must be satisfied

19) $$\left\lceil \frac{3 N_v + 3}{j - 3} \right\rceil > \left\lceil (\alpha - \beta) N_v \right\rceil - 1 \quad .$$

Proof: Note the $Alg(\alpha, \beta, K, j)$ mainline will apply the PENALTY-P subroutine to node v only if this node's $p(v)$ value lies outside the $[\beta, 1 - \beta]$ range. Also, note that Lemma 3.6.D implies that the period between the last time when $p(v)$ retained a value inside the $(\alpha, 1 - \alpha)$ interval and the present moment when it moves outside the $[\beta, 1 - \beta]$ interval must include a minimum of $\left\lceil (\alpha - \beta) N_v \right\rceil$ insertion and deletion commands which are applied to v's set of leaf descendants. This fact, in other words, means that there must be a minimum of $\left\lceil (\alpha - \beta) N_v \right\rceil - 1$ commands that are applied to v's subtree during the interim period when $p(v)$ lies in either the $[\beta, \alpha]$ or $[1 - \alpha, 1 - \beta]$ intervals. The proof of the present theorem can be completed if we observe that Lemma 3.6.G implies that this same set of insertion and deletion command is forbidden from containing more than $\left\lceil \frac{3 N_v + 3}{j - 3} \right\rceil$ instructions. The observations in the preceeding two sentences imply that equation 19 must hold.

Q. E. D.

THEOREM 3.6.I. Let Δ, α, β, K, j, T, c, v and N_v have the same meaning as they had in the previous propositions. Let us assume that the

parameter of j satisfies both the previous equations of 11 and 12. The latter of these two equations was only briefly mentioned in our earlier discussion, and it is therefore reproduced below

20) $$j > 3 + \frac{3}{\alpha - \beta} \quad .$$

Under these circumstances, there will exist a constant of M such that

i) The $Alg(\alpha, \beta, K, j)$ mainline will apply the PENALTY-P subroutine to the subtree whose root is v only if N_v is less than or equal to M at the time of this subroutine-call.

ii) And all such subroutine-calls will be guaranteed to consume $\Theta(w)$ worst-case runtime.

Proof of Proposition I. It is easy to verify that equation 20 implies that there must exist a number of M such that at' $_v > M$ will fail to satisfy equation 19 of Theorem 3.6.II. This fact together with the cited theorem implies that no subroutine-call will be made to the PENALTY-P subroutine when N_v has such a high value.

Q. E. D.

Proof of Proposition II. This proposition is a consequence of Lemma 3.5.G and Proposition I. The cited lemma indicated that the PENALTY-P subroutine will operate in $\Theta(w N_v \log N_v)$ runtime, and Proposition I imposed a bound on the allowed value of N_v. The combined implication of these two

assertions is that the PENALTY-P subroutine is guaranteed to operate in $\Theta(w\ M \log M)$ runtime. The remaining aspect of the proof of Proposition II is based on the observation that M should be regarded as a constant since its value is unrelated to that integer of N which indicates the size of our super-B-tree. (Instead, the value of constant M is only a function of other fixed parameter constants such as Δ, α, β, K and J.) It thus follows that the M log M component of the $\Theta(M \log M\ w)$ runtime should be regarded as a coefficient of a runtime that has an $\Theta(w)$ magnitude.

Comment. One further remark should be made about this M log M runtime coefficient. It is easy to verify that there do exist K, J and Δ parameters which will cause M to assume a value of $\frac{1}{\alpha - \beta}$. In such cases, the M log M coefficient will have a very small size.

The next theorem will tie together all the preceeding results and prove that it is possible to develop a super-B-tree algorithm that operates in $\Theta(w \log N)$ worst-case runtime. It should be stated that the main parts of the proof of the super-B-tree theorem were discussed in the last few propositions. This is because the PENALTY-P subroutine is the only aspect of the $Alg(\alpha, \beta, K, J)$ procedure that is capable of consuming prohibitive amounts of runtime. Thus, the importance of the last several propositions is that they have performed the most complicated aspects of the needed runtime analysis by showing how the proper set of α, β, K, J and Δ parameters will adequately

restrain the runtime of the PENALTY-P subroutine.

THEOREM 3.6.J. Let us assume that the Δ, α, β, and K parameters satisfy claim 3.6.A and that the parameter of J satisfies the previous equations 11 and 12. Let T denote a super-B-tree, and let us assume that all insertions and deletions in tree T are performed by the $Alg(\alpha, \beta, K, J)$ procedure. Under these circumstances it will follow that

I) The $Alg(\alpha, \beta, K, J)$ procedure will correctly perform those insertion and deletion operations which are indicated by the user.

II) This procedure will guarantee that tree T has an $\Theta(\log N)$ height.

III) The $Alg(\alpha, \beta, K, J)$ procedure will perform insertion and deletion operations in $\Theta(w \log N)$ worst-case runtime.

Proof: The first two propositions are basically a restatement of earlier results that were proven in Propositions 3.5.H and 3.6.C. Thus, only Proposition III requires proof to complete the verification of the present theorem. This proof will be based on the following four observations:

I) It is trivial to verify that step 1 of $Alg(\alpha, \beta, K, J)$ will operate in $\Theta(\log N)$ runtime (since Proposition II showed that T had an $\Theta(\log N)$ height).

2) Lemma 3.6.D implies that step 2 will operate in $\Theta(w \log N)$ runtime (because tree T has an $\Theta(\log N)$ height).

3) Proposition 3.4.1, 3.5.F and 3.6.I respectively imply that a single invocation of substeps 3a, 3b, and 3c will have $\Theta(w)$ worst-case runtimes. (The coefficients of these runtimes will of course be functions of the five fixed parameters of α, β, K, J and Δ. These coefficients were not included in my estimates of runtime orders of magnitudes because the size of those five numbers are unrelated to the size, N, of our super-B-tree or to the argument of w.)

4) There will be no more than $\Theta(\log N)$ invocations of the preceeding three substeps of 3a, 3b and 3c (because Proposition II indicated that tree T had an $\Theta(\log N)$ height).

The combined implication of these four observations is that all steps of the $Alg(\alpha, \beta, K, J)$ procedure will consume a total of no more than $\Theta(w \log N)$ worst-case runtime.

Q.E.D.

The previous theorems and proofs clearly demonstrated how the proper set of Δ, α, β, K and J parameters would cause the $Alg(\alpha, \beta, K, J)$ procedure to operate in the desired $\Theta(w \log N)$ worst-case runtime. That discussion was predicated on the assumption that there do, in fact, exist parameters of Δ, α, β, and K which satisfy the conditions of claim 3.6.A.

The last goal of this chapter will be to verify this fact. The discussion of these parameters will begin with the following preliminary lemma.

LEMMA 3.6.K. Let $Q(K)$ denote the same quadratic which it has previously represented (this quadratic is reproduced below in equation 21), and let α and K denote two numbers which satisfy the inequalities that are shown in equations 22 and 23:

21) $$Q(K) = 1 - \frac{\sqrt{K^2 - K}}{K}$$

22) $$K > 2$$

23) $$0 < \alpha < Q(K) \quad .$$

Under these circumstances, there will exist a number of d which satisfies the inequality of

24) $$d > \alpha$$

and which additionally satisfies the condition that if $p(v) = 1 - \alpha$, if every son and grandson, s, of v has a $p(s)$ value that belongs to the $[\alpha, 1 - \alpha]$ interval, and if the DETERMINE(α, K) subroutine (which was formally described in part 3.2.K of this chapter) recommends that node v should be rebalanced with rotation R then this rotation will sufficiently modify the value of $p(v)$ to insure that $p(v)$ belongs to the $[d, 1 - d]$ interval and to insure that every renewal node, n, that is produced by rotation R

will similarly have a $p(n)$ value that belongs to the same $[d, 1-d]$ range.

Proof: Let $a_1 a_2 \cdots a_5$ and $b_1 b_2 \cdots b_5$ denote those numbers which are defined in the following ten equations:

25) $$a_1 = \alpha(2\alpha - 1)$$

26) $$b_1 = 1 - (K - 1)\alpha$$

27) $$a_2 = \frac{\alpha}{1 - (K - 1)\alpha}$$

28) $$b_2 = \frac{1}{2 - \alpha}$$

29) $$a_3 = \alpha(2 - K\alpha)$$

30) $$b_3 = \alpha + (1 - \alpha)^3$$

31) $$a_4 = \frac{\alpha}{\alpha + (1 - \alpha)^3}$$

32) $$b_4 = \frac{1}{2 - K\alpha}$$

33) $$a_5 = \frac{1 - K\alpha}{K - K\alpha}$$

34) $$b_5 = \frac{(1 - \alpha)^2}{(1 - \alpha)^2 + \alpha} \quad .$$

An examination of the DETERMINE(α, K) subroutine reveals that it will always recommend that either an R_1 or R_2 rotation be performed when it is instructed to examine a node v whose $p(v)$ value equals $1 - \alpha$. In

this proof, it will be assumed that v_A, v_B and v_C will denote the three renewal nodes which the previous diagrams of 3.2.D and 3.2.E indicated would be produced by the R_1 and R_2 rotations. A straightforward algebraic analysis reveals that the DETERMINE(α, K) procedure (which was described in part 3.2.K of this chapter) will recommend the R_1 rotation only if this transformation will produce nodes that satisfy the inequalities of:

35) $$a_1 \le p(v) \le b_1$$

36) $$a_2 \le p(v_A) \le b_2$$

and that the R_2 rotation will be recommended by the DETERMINE(α, K) subroutine only if the latter transformation produces nodes which satisfy:

37) $$a_3 \le p(v) \le b_3$$

38) $$a_4 \le p(v_B) \le b_4$$

39) $$a_5 \le p(v_C) \le b_5 \quad .$$

Let d denote that number which is produced by taking the minimum value that is begotten by the five a_i number together with the five additional numbers of the form $(1 - b_i)$. Equations 21, 22 and 23 and equations 25 through 34 can be shown to imply that $d > \alpha$. Also, this fact, the definition of d, and equations 35 through 39 will imply that the DETERMINE(α, K) subroutine will recommend a rotation whose node v and whose renewal nodes will necessarily be assigned p-values that will lie in the $[d, 1-d]$ range.

These observations are important because they show that it is possible to construct a number of d which satisfies the conditions of this lemma. Q. E. D.

The preceeding lemma is very important because it will be used in the proof of claim 3.6.A. The statement of this claim was originally given at the beginning of this section, and it is reproduced below:

Claim 3.6.A. There will exist numbers of Δ, α, β, and K such that the first three of these numbers will satisfy the condition of

40) $$1/3 > \Delta > \alpha > \beta > 0$$

and such that these four numbers will satisfy the additional condition that if v is a node whose $p(v)$ value lies inside either the $[\beta, \alpha]$ or $[1-\alpha, 1-\beta]$ intervals, if additionally all the sons and grandsons, s, of v will satisfy the condition that their $p(s)$ values lie inside the $[\beta, 1-\beta]$ interval, and if the DETERMINE(α, K) subroutine recommends that rotation R be used to rebalance node v then this rotation will

i) sufficiently modify the value of $p(v)$ to insure that it lies inside the $(\Delta, 1-\Delta)$ interval;

ii) and this rotation will produce renewal nodes of s^* whose $p(s^*)$ values also lie in the same $(\Delta, 1-\Delta)$ range.

Proof: Let d, α, K denote three numbers that satisfy the preceeding Lemma 3.6.K, β denote an arbitrary number that is less than or equal to α,

T denote a subtree whose root of v satisfies $p(v) \geq 1-\alpha$, and R denote that rotation which the DETERMINE(α, K) subroutine recommends for node v. Note that the application of rotation R to tree T will change the value of $p(v)$ and that it will produce one or two renewal (which shall be denoted as s_1^* and s_2^*). In this proof, $F(T, \alpha, K)$ will denote the minimum of the six values of $p(v)$, $1-p(v)$, $p(s_1^*)$, $1-p(s_1^*)$, $p(s_2^*)$ and $1-p(s_2^*)$ that this rotation will produce on tree T. Let $z(\beta, \alpha)$ denote the set of all possible trees which (before the application of rotation R) have a root whose $p(v)$ value belongs to the $[1-\alpha, 1-\beta]$ interval and which satisfies the additonal condition that if s denotes any of the sons or grandsons of v then $p(s)$ must belong to the $[\beta, 1-\beta]$ interval. Let $F(\beta, \alpha, K)$ denote the minimum $F(T, \alpha, K)$ value for all trees T that belong to this $z(\beta, \alpha)$ set. The $F(\beta, \alpha, K)$ function will be very important, and the next paragraph will explain how it will enable us to prove Claim 3.6.A.

This proof will be based on the following two observations:

A) Lemma 3.6.K and the definition of the $F(\beta, \alpha, K)$ function easily imply that $F(\alpha, \alpha, K) \geq d$.

B) The $F(\beta, \alpha, K)$ function can be easily shown to be continuous in the variable of β.

The preceeding two observations imply that $F(\beta, \alpha, K) > \frac{d+\alpha}{2}$ when β is chosen to be sufficiently close to α. The remainder of the proof is quite

simple. Let β be some number that is less than α and satisfies the previous condition. Let us set Δ equal to the number of $\frac{d+\alpha}{2}$. Under these circumstances, it must follow that $F(\beta, \alpha, K) > \Delta$, and this will in turn imply that conditions I) and II) of claim 3.6.A must hold for every subtree T whose $p(v)$ value initially belongs to the $[1-\alpha, 1-\beta]$ interval. The alternate case where $p(v)$ belongs to the $[\beta, \alpha]$ interval will of course follow from mirror image symmetry arguments. Claim 3.6.A has thus been verified since this proof has shown that there exists parameters of Δ, α, β, and K which satisfy its conditions I) and II) for all the required trees.

Q. E. D.

The preceeding claim 3.6.A is clearly very important since all the theorems in this chapter were predicated on the assumption that this claim was correct. The above proof has thus completed this chapter's discussion of *super-B-trees* by verifying this assumption.

In the rest of this thesis, any version of the $Alg(\alpha, \beta, K, j)$ procedure whose parameters will satisfy the preceeding claim 3.6.A as well as equations 11 and 12 will be called a "*super-B-tree algorithm.*" Such a version of the $Alg(\alpha, \beta, K, j)$ will be very important because Theorem 3.6.J has shown that it will operate in $\Theta(w \log N)$ worst-case runtime.

There are many examples of useful applications of super-B-trees. For example, consider the predicate shown below in equation 41. Knuth has stated that this predicate is an example of an expression that no one has

previously efficiently solved (KNUTH-73).

41) $e(x, y) = \{x.a_1 < y.b < x.a_2 \text{ AND } x.a_3 < y.b_2 < x.a_4\}$.

Chapter 6 will show how this predicate can be efficiently solved with a super-B-tree whose two fundamental characteristics are that its y-leaves are arranged by order of increasing $y.b_1$ value and that its SDS fields consist of traditional B-trees whose leaves are arranged by order of *increasing* $y.b_2$ value. The specific results with regard to Knuth's predicate will be that if $\bar{x}$ denotes a fixed element and if $I_e(\bar{x})$ denotes the number of y-records that satisfy $e(\bar{x}, y)$ then the preceeding super-B-tree will enable us to locate the full set of $I_e(\bar{x})$ distinct y-records that satisfy $e(\bar{x}, y)$ in $\Theta(I_e(x) + \log^2 N : \log^2 N)$ worst-case runtime. Such a solution of Knuth's unsolved predicate will clearly possess an efficient data-retrieval and data-modification time. It is also easy to see that such a super-B-tree will occupy $\Theta(N \log N)$ memory space. This super-B-tree will thus also be efficient from the perspective of memory utilization.

The detailed description of how super-B-trees may be applied to all types of E-7 expression will be given in Chapter 6. It should be stated that some reader may prefer to immediately proceed to that chapter because there will be no further mention of super-B-trees in chapters 4 and 5. The subject matter in this thesis has been carefully organized so that the reader can skip most of chapters 4 and 5 provided he is acquainted with the introductory

parts of these chapters (which are sections 4.1 and 5.1). Some readers may prefer to follow that course so that they can more quickly learn about the many applications of super-B-trees.

Remark 3.6.L. Finally, it should be stated that I do not recommend that a super-B-tree algorithm be implemented in the literal manner that has been outlined in this chapter. This is because the runtime coefficients of the super-B-tree algorithm has been deliberately ignored in this chapter for the sake of simplifying the proof of the super-B-tree theorem. A commercial implementer would of course care little about simple proofs and would instead want that version of the super-B-tree algorithm which has the lowest possible runtime coefficient for performing its $\Theta(w \log N)$ operations. There are several additional modules which should be added to a super-B-tree algorithm for the purposes of improving its runtime coefficient. Some useful coefficient optimization techniques are listed below:

1) The GCAS subroutine will be more efficiently employed if its argument of j consists of a variable that is designed to increase as $p(v)$ moves away from $1/2$ rather than the fixed constant which was mentioned in this chapter.

2) Let v_C and v_C' denote those members of the ANTICIP(v) set that were previously illustrated in the familiar Diagram 3.2.E and 3.2.G. It is fairly easy to show that these two anticipated

nodes will also be members of the anticipated sets of v's sons. The coefficient of the $Alg(\alpha, \beta, K, j)$ procedure will be improved if this fact is taken into consideration and if the procedure is designed to insure that node v and its sons do not possess redundant repetitions of the same anticipated SDS field.

3) Let us assume that R denotes a rotation whose application to node v causes the leaf-depths of tree T to decrease. Under these circumstances, the runtime coefficient of the $Alg(\alpha, \beta, K, j)$ procedure will be further improved if it employs a module that automatically applies such a rotation of R to node v whenever the SDS fields of the members of the ANTICIP(v, R) set have reached fully constructed states.

4) The runtime coefficient of the GCAS subroutine will be improved if this procedure is revised so that it only inserts new y-leaves into the $SDS(v_A)$, $SDS(v_B)$ and $SDS(v_C)$ fields when $p(v) > 1/2$, and only inserts y-leaves into $SDS(v_A')$, $SDS(v_B')$, $SDS(v_C')$ when $p(v) < 1/2$. Also, the requirements of conservation of memory-space will usually be best served if the first three SDS fields are reduced to starting states when $p(v) < 1/2$ and the latter three are similarly reduced to starting

states when $p(v) > 1/2$.

5) The super-B-tree algorithm should be designed to distinguish those times when the computer is mostly busy from the other times when it is mostly idle. As a general rule, this algorithm should encourage the GCAS and PENALTY-P subroutines to perform most of their work when the computer is less than very busy. Also, it will be desirable to completely reorganizae the super-B-tree for the purposes of giving it nodes whose $p(v)$ values are as close as possible to $1/2$ during large periods of times when the computer would otherwise be idle (such as on week-ends).

6) The super-B-tree algorithm should very carefully choose its α, β, K and J parameters. Often, one choice of parameters will produce a better coefficient for the $\Theta(w \log N)$ runtime than another set of values. The super-B-tree algorithm should attempt to find the optimal α, β, K and J parameters by considering the preceeding five optimizations as well as the cost of memory, the cost of runtime, the frequency of insertions, the frequency of deletions, the frequency of retrievals, and the trees expected leaf depth.

The discussion in this remark clearly contained a very brief summary of the

techniques that can further improve the runtime coefficient of this chapter's super-B-tree algorithm. This discussion was kept brief because it would have taken me too much time to write more details about these coefficient optimization techniques. It should also be stated that the Alg(α, β, K, J) procedure will usually be quite efficient even when none of these coefficient optimization techniques are applied.

CHAPTER 4

4.1 Chapter Overview

The "E-3 subclass" of E-7 expressions will be the main subject of the next two chapters. This subclass will be defined to be the subset of E-7 expression whose every atomic predicate is an equality predicate. Examples of E-3 expressions are given in equations 1 through 6 below:

1) $\{x.a = y.b\}$

2) $\{x.a_1 = y.b_1 \text{ OR } x.a_2 = y.b_2\}$

3) $\{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2\}$

4) $\{\text{NOT } x.a = y.b\}$

5) $\{x.a_1 = y.b_1 \text{ OR } [x.a_2 = y.b_2 \text{ AND NOT } x.a_3 = y.b_3]\}$

6) $\{x.a_1 = y.b_1 \text{ AND NOT } [x.a_2 = y.b_2 \text{ OR } x.a_3 = y.b_3]\}$.

It will be shown in this chapter that SUMs and COUNTs for E-3 expressions can be calculated in $\Theta(1;1)$ time. A similar result for FIND operations on E-3 expressions will be discussed in the next chapter. Both these results are important because they are useful in their own right and because the E-3 databases will be used as SDS structures during chapter 6's discussion of E-7 expressions.

The discussion in this chapter will be divided into four main parts. The

hashing algorithms that will be used in both this chapter and the next chapter will be defined in section 4.2. The later parts of this chapter will consist of a three part description of the SUM-3 algorithm. That discussion will begin with an initial description of the SUM-1 algorithm (in section 4.3), and it will subsequently turn to the further generalized SUM-2 and SUM-3 algorithms (in Section 3.4).

Throughout this chapter, the symbol $S_e^F(\bar{x})$ will denote the sum of F(y) values of those y-records (in R_y) which satisfy $e(\bar{x}, y)$. The distinction between the three SUM algorithm will be that they will calculate the $S_e^F(\bar{x})$ values for threee progressively broader classes of E-3 expressions. The SUM-1 algorithm will process only a fairly simple subclass of predicate called "E-1 expressions." The SUM-2 and SUM-3 algorithm shall be designed to process the progressively more sophisticated "E-2" and "E-3 predicates". All the theorems and lemmas in this chapter are intended to help eventually prove that SUM-3 algorithm can process all E-3 predicates in $\Theta(1;1)$ worst-case runtime.

The subject matter in this chapter is so simple that it can almost be categorized as trivial. This discussion has been included in this thesis primarily because the SUM-3 and COUNT-3 algorithms will be used as subroutines by the more advanced algorithms of the next two chapters. A second reason for the inclusion of this chapter is that it will provide a fairly

simple setting for introducing the general methods of reasoning that will be used throughout this thesis.

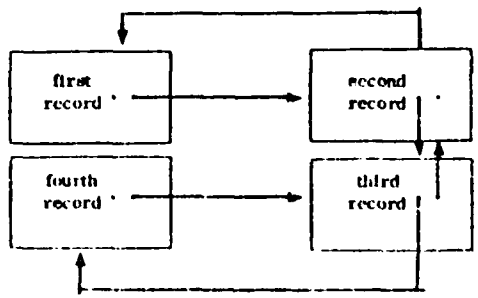
It should be stated that it is not necessary for the reader to examine this chapter if he accepts that it is possible to calculate SUMs and COUNTs for E-3 expressions in $\Theta(1:1)$ worst-case hashtime. Some readers may prefer to skim or omit this chapter so that they can more rapidly approach the more advanced and genuinely subtle algorithms of the later chapters.

4.2 Hashing

There will be three kinds of hash algorithms discussed in this thesis. These will be the simple hashlist, the hash-pointer-structure, and the additive hash. A precise description of these three concepts will be given in this section. Most of the discussion of hashing will be fairly simple. My recommendation is that the reader skim this section on his first reading of this thesis.

All of the hashing algorithms in this thesis will employ the traditional hash-random addressing system for locating computer records in the designated memory file. It is well known that such hashing strategies insert, delete or locate a given record in (stochastically) expected $\Theta(1)$ time (AHU-74). A special list structure will be assigned to the hash files in this thesis. This list-structure will be generated by a pair of forward and back-pointers that is assigned to each record in our hashed file. These pointers will cause the hashed records to form a "chained list" where the records are enumerated in the order that they were initially inserted into the hashed file (as shown in Diagram A).

DIAGRAM A:



The relevant hash algorithms will obviously update these pointers in the straightforward manner when the user indicates that a given record should either be inserted into or deleted from the hashfile. It is fairly trivial to verify that such pointers do not change the basic $\Theta(1)$ runtime of the hash algorithms.

The hashing algorithms in this thesis will be applied in a slightly different context from the traditional usage of hashing. The distinction is that traditional hash algorithms presupposed that the user was actively watching the hash algorithm and deciding how much memory should be allocated to optimize performance. The hash algorithms in this thesis will not require the user to play such an active role. These algorithms will thus be designed to make their own optimizations and automatically decide how much memory should be allocated.

A partial description of a hash algorithm that manages its own memory was given in AHU-74. Aho, Hopcroft, and Ullman did not indicate who was the original author of their textbook hash algorithm or give it a name. In this thesis, their hash algorithm will be called the AHU hash procedure. The AHU algorithm caused the hash files to attain a size which was always a power of 2. This procedure automatically increased the size of hashed memory file by a factor of 2 whenever the hashed file became more than half-filled. The proposed hashed file would thus always be between 25% and 50% filled. Aho, Hopcroft, and Ullman chose files with such low densities because they supported rapid hash-searches with approximately 2 or less expected units of runtime.

The AHU hashing algorithm clearly has many desirable features, and a somewhat similar algorithm will be used in this thesis. The hashing algorithm in this thesis will not be identical to the AHU algorithm because Aho, Hopcroft, and Ullman did not intend for users to execute their algorithm in the literal manner suggested by their textbook. The difficulty with the AHU algorithm can be understood if we consider the case where the insertion of an additional record causes the hashfile to exceed its 50% density-threshold. In that case, a literal interpretation of the AHU algorithm would suggest that this procedure would spend $\Theta(N)$ time reorganizing its hashfiles. Such an expenditure of $\Theta(N)$ time is clearly undesirable because it would violate the proclaimed $\Theta(1)$ worst-case-expected runtime of hash algorithms. It will thus be necessary to develop a slightly more efficient hash algorithm for the purposes of this thesis.

The hash algorithm in this thesis will be based on a principle of gradually reorganizing the hashfiles rather than performing entire file reorganizations in one expensive $\Theta(N)$ step. The gradual procedure will thus slowly reorganize the hashfile during the period when file-density ranges between 45% and 50%. The gradual reorganization procedure will be discussed in detail in Propositions 3.2.A through 3.2.D. Its basic advantage is that it is much more stable and predictable than the AHU reorganization procedure. Operations under the gradual-hash will thus always consume $\Theta(1)$ expected-time (and only the runtime coefficients of these operations will increase during the

reorganization period). My recommendation is that the reader skim the next three propositions on his first reading of this thesis since gradual-hashing is only peripherally related to the main topics of this thesis.

LEMMA 4.3.A. It is possible to design a "gradual-hash-algorithm" which will have an $\Theta(1)$ worst-case-expected runtime, and which will double the size of its hashfiles during the transition period when hash density falls between 45% and 50%.

Proof: In this proof, a "gradual-hash data-modification procedure" will be described that satisfies the conditions of this lemma. More specifically, the proposed algorithm will be shown to:

i) correctly modify the data base in accordance with the user's insertion and deletion commands;

ii) simultaneously expand the hash memory so that it will double in size during the 45-50% transition period;

iii) and execute the user's specified insertion or deletion commands in $\Theta(1)$ worst-case-expected time.

The goal of this proof will be to show how insertion and deletion algorithms can be designed that satisfy the preceeding conditions. This discussion will begin with a description of the needed insertion algorithm.

The insertion module of the gradual hash procedure will be designed in the more or less obvious manner. The proposed insertion procedure will have two steps. The first step will add the specified record to the prevailing hash-files. The second part of the insertion procedure will check to see whether the old hashfile has a density varying between 45% and 50%. If this is so then the insertion algorithm will essentially take ten additional elements from the old hashfile and make replica-copies of them in the new file that is being constructed. (The ten elements in the old hashfile will be located via a straightforward scan of the previously described "chained hashlist.") Clearly the just described insertion algorithm will construct the needed new hashfile in the desired time before the 50% transition-density is reached (since ten records are added to the new file whenever one record is added to the old file).

The transition-algorithm will execute the user's deletion commands in a similar straightforward manner. The transition algorithm will respond to the user's deletion requests with a two-part procedure. The first part of this procedure will delete the given record from the old hashfile. The second part will check to see whether the same data-record is also stored in the new hash-file. If this data-record is present in the new file, it will obviously also be deleted.

It is easy to confirm that the above described transition-hash-insertion and transition-hash-deletion algorithms will manipulate the database in the correct manner declared by claims i) through iii). Thus, it has been proven

that it is possible to design a hash algorithm that performs the desired task of doubling the hashfile storage without harming its basic $\Theta(1)$ insertion and deletion operations.

Q. E. D.

Observation 4. 2. B. It is important to note that the gradual-hash algorithm will require approximately the same total time as the AHU procedure for transforming an old hashfile into the new expanded hashfile. The advantage to the gradual-hash procedure is that it will extend this file transformation over a large number of insertion and deletion operations. Such a gradual transformation is desirable because most users will prefer an $\Theta(1)$ hash procedure with a slightly higher runtime coefficient over a less predictable procedure which will occasionally consume prohibitive $\Theta(N)$ time.

COROLLARY 4. 2. C. It is also possible to design a gradual-hash-algorithm which will have an $\Theta(1)$ worst-case-expected runtime, and which will cut the storage space in half during the transition-period when file density falls between 25% and 22.5%.

Proof: The hash procedure suggested by Corollary 4. 2. C is essentially the straightforward inverse of the Lemma 4. 2. A procedure. Thus the preceeding proof also applies to the present corollary.

Q. E. D.

COROLLARY 4. 2. D. It is possible to design a hash-algorithm which will perform insertion, deletion, and search operations in $\Theta(1)$ worst-case-expected

time, and whose file densities will always vary between 22.5% and 50%.

Q. E. D.

Proof: Clearly the preceeding two propositions imply that such an algorithm is possible to design.

Q. E. D.

Remark 4. 2. E. There are several further optimizations which can be added to the gradual-hash algorithm for the purposes of somewhat improving the runtime coefficient. For example, it would be desirable for a hash-algorithm to distinguish between the periods when the computer is mostly busy as opposed to when it is mostly idle. Obviously, a further optimized algorithm would attempt to postpone the task of changing the size of the hashfiles to those periods when the computer is idle. There are also other optimizations which could have been discussed in this section. These optimizations were not discussed for two reasons. The first is that the present hash-algorithm is clearly adequately efficient by itself (since it operates on $\Theta(1)$ time). Also, the topic of further optimizing the runtime coefficient was omitted in order to keep the discussion brief.

All the hashing algorithms in this thesis will employ the techniques that were outlined in Theorems 4. 2. A through 4. 2. D. There will be three specific hash-algorithms that will be discussed in this thesis. There will be the hashed-list, the hashed-pointer-structure, and the additive hash. The first of these hash algorithms requires no further explanation since the hashed-list was essentially the data structure that has been so far described in this section. The hashed-

pointer-structure also requires very little explanation. This structure is simply defined to be a special hashed-list whose data-records are pointers.

The third type of hashed file discussed in this thesis will be the additive hash. In the discussion of this file, it will be assumed that $S^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$ denotes the sum of the $F(y)$ values of those y-records (in R_y) which satisfy the condition:

1) $$y.b_1 = c_1 \text{ AND } y.b_2 = c_2 \text{ AND} \cdots \text{ AND } y.b_j = c_j \quad .$$

The definition, below, will use this $S^F_{b_1 b_2 b_j}(c_1 c_2 \cdots c_j)$ sum to define the additive hash:

<u>Definition 4.2.F.</u> A hash function, denoted as $H^F_{b_1 b_2 \cdots b_j}$, which maps "$c_1 c_2 \cdots c_j$" tuples onto $H^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$ records will be said to be <u>an additive hash that "describes" function</u> F <u>and attribute list</u> $b_1 b_2 \cdots b_j$ if every such $c_1 c_2 \cdots c_j$ tuple satisfies the following two conditions:

1) The $H^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$ record will exist in the hashfile if and only if $S^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j) \neq 0$.

2) When the preceeding condition is satisfied, the $H^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$ record will contain a number indicating the value of $S^F_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$.

<u>Example 4.2.G.</u> Suppose the R_y relation contains the four y-records indicated by the table below:

element-name	$y.b_1$ value	$y.b_2$ value	F(y)-value
$\overline{y_1}$	1	1	3.1
$\overline{y_2}$	1	1	3.2
$\overline{y_3}$	1	2	3.3
$\overline{y_4}$	2	1	3.4

Obviously under these circumstances, it will follow that $S^F_{b_1 b_2}(1, 1) = 6.3$, $S^F_{b_1 b_2}(1, 2) = 3.3$, and $S^F_{b_1 b_2}(2, 1) = 3.4$. The additive hash will thus store these three numbers in the respective locations of $H^F_{b_1 b_2}(1, 1)$, $H^F_{b_1 b_2}(1, 2)$ and $H^F_{b_1 b_2}(2, 1)$. The additive hash will not contain any other additional records because the $S^F_{b_1 b_2}$ values of all other ordered pairs equal zero.

The additive hash is an important concept which will have major applications throughout this chapter.

4.3 The SUM-1 Algorithm

A predicate, $e(x, y)$, will be said to be an E-1 predicate if it is either the trivial predicate or "TRUE" or a conjunction of several xy-equality predicates, similar to equation 1:

1) $$x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2 \text{ AND} \cdots x.a_j = y.b_j \quad .$$

The E-1 expressions are obviously a subclass of the broader category of E-3 expressions.

Let $S_e^F(\bar{x})$ once again denote the sum of the $F(y)$ values of those y-records (in R_y) which satisfy $e(\bar{x}, y)$. In this section, it will be shown that the SUM-1 algorithm can calculate the $S_e^F(\bar{x})$ totals for E-1 expressions in highly optimal $\Theta(1;1)$ runtime.

The discussion of the SUM-1 algorithm will be divided into three parts that will discuss the compiler, data-retrieval module, and data-modification module of this algorithm. These three modules will perform the standard three tasks which jointly constitute a full database procedure. Thus the SUM-1 compiler will design the database, the SUM-1 retrieval module will search this database (whenever the user asks for a $S_e^F(\bar{x})$ value,) and the modification module will update this database in accordance with the user's insertion and deletion commands.

The additive hash (described in Definition 4.2.F) will constitute the

basic data-structure used by the SUM-1 algorithm. In the discussion which follows, it will be assumed that $e(\bar{x}, y)$ denotes a general E-1 expression, similar to equation 2 below:

2) $e(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2 \text{ AND} \cdots x.a_j = y.b_j\}$.

Throughout this discussion, the list of b_j-symbols appearing in predicate e will be called the y-attributes of e. Thus, the list of equation 2's y-attributes will simply be: $b_1 b_2 \cdots b_j$.

The SUM-1 compiler will employ the y-attributes of equation 2 in the more or less obvious manner. The algorithm will be initially activated when the user supplies it with a predicate e, a function F, and informs it that he wishes the algorithm to develop the capacity to calculate the S_e^F sums with maximum efficiency. The SUM 1 compiler will respond by creating an additive hash file whose associated function is F and whose associated attribute-list is e's y-attributes. In the example of equation 2, the produced hash file is simply the $H^F_{b_1 b_2 \cdots b_j}$ additive hash. It is easy to verify that if R_y is a relation with N distinct elements then the amount of time needed to construct this hash-file is simply $\Theta(N)$ time (and similarly the hash will occupy approximately $\Theta(N)$ storage space).

The SUM-1 retrieval module will use this additive hash in the more or less obvious manner when answering the user's data-retrieval requests. For example, if the user asks the SUM-1 algorithm to calculate the value of

$S_e^F(\bar{x})$ then it will respond by taking the tuple of "$\bar{x}.a_1, \bar{x}.a_2, \cdots, \bar{x}.a_j$" and hashing to the associated address of $H^F_{b_1 b_2 \cdots b_j}(\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_j)$. If an $S^F_{b_1 b_2 \cdots b_j}(\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_j)$ number is stored at this address, then the SUM-1 algorithm will inform the the user that $S_e^F(\bar{x})$ equals that quantity. If, on the other hand, no $S^F_{b_1 b_2 \cdots b_j}(\bar{x}.a_1, \bar{x}.a_2, \cdots \bar{x}.a_j)$ number is found at the calculated hash address then the SUM-1 algorithm will inform the user that $S_e^F(\bar{x})$ equals zero. It is an obvious consequence of the definition of additive hashing that the preceeding procedure will correctly calculate the value of $S_e^F(\bar{x})$ for the specified E-1 predicates, and that it will also perform this task in $\Theta(1)$ runtime.

The data-modification module of the SUM-1 algorithm will also operate in the obvious manner. Thus if the user instructs this algorithm to insert a record, $\bar{y}$, into the database then the algorithm will respond by taking the j-tuple of "$\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j$" and hashing to the associated address of $H^F_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$. If an $S^F_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ number is found at this address then the SUM-1 algorithm will increment this quality by an amount of $F(\bar{y})$ (so that the revised database will reflect the fact that $\bar{y}$ has been inserted into it). The data-modification module will have a very different response when there is no $S^F_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ number stored at the address of

$H^F_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$. In that case, the data-modification module will create a new record and set its initial sum-value equal to $F(\bar{y})$ (since in this case $\bar{y}$ is the only record which the $S^F_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2, \cdots \bar{y}.b_j)$ sum should reflect). It is easy to confirm that the just-proposed procedure will correctly perform insertions in the required $\Theta(1)$ runtime. Deletion operations are of course performed by using the straightforward inverse of the preceeding insertion algorithm.

The discussion in this section was so simple that no formal proof is needed to verify the correctness of the SUM-1 algorithm or to confirm its ability to operate within the designated $\Theta(1;1)$ runtime bounds. All these facts are an obvious consequence of the definition of the SUM-1 algorithm, the definition of additive hashing, and the statement of Corollary 4.2.D (which asserted the efficiency of all the hash algorithms in this thesis). Thus, the preceeding discussion is summarized by the following theorem:

<u>THEOREM 4.3.A.</u> A sum calculation algorithm (called "SUM-1") can be designed which will efficiently calculate the $S^F_e(\bar{x})$ value for E-1 expressions in accordance with the following 3 conditions:

1) The database for performing the S^F_e calculations will require $\Theta(N)$ storage space and $\Theta(N)$ time to construct. (This database is simply the $H^F_{b_1 b_2 \cdots b_j}$ additive-hash data structure.)

2) After the above database is constructed, the SUM-1 algorithm can calculate the value of $S^F_e(\bar{x})$ in $\Theta(1)$ runtime.

3) The SUM-1 algorithm will need only $\Theta(1)$ runtime to modify this database when the user gives an insertion and deletion command.

<u>Example 4.3.B.</u> Let $e(x, y)$ denote the predicate given in equation 3, let $\bar{x}_1$ and $\bar{x}_2$ denote the respective ordered pairs of $(1, 1)$ and $(2, 2)$, and let R_y denote the same relation as was described in Example 4.2.F.

3) $$e(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2\} \quad .$$

In this example, we will watch the SUM-1 algorithm calculate $S^F_e(\bar{x})$ values and interact with our $H^F_{b_1 b_2}$ database.

The mechanics of the SUM-1 algorithm are fairly simple. Suppose the user instructed the algorithm to calculate the value of $S^F_e(\bar{x}_1)$. In this case, the data-retrieval module would hash on the key of "1, 1" and find the record that has previously been denoted as $H^F_{b_1 b_2}(1, 1)$. Note that the number 6.3 is stored in this record (as was shown in Example 4.2.F). It thus follows that the SUM-1 algorithm will answer the user's request by informing him that $S^F_e(\bar{x}_1)$ equals 6.3.

As a second example, suppose the user asked the SUM-1 algorithm to calculate $S^F_e(\bar{x}_2)$. Note that the additive-hash contains no record describing

the ordered pair of "2, 2" in its memory. The SUM-1 algorithm will therefore inform the user that $S_e^F(\bar{x}_2)$ equals zero.

Thirdly, let us suppose that the user instructs the SUM-1 algorithm to insert a new record into the database. We will denote this record as $\bar{y}_5$, and we will assume that $\bar{y}_5 . b_1 = 1$, $\bar{y}_5 . b_2 = 1$, and $F(\bar{y}_5) = 3.5$. In this case, the SUM-1 algorithm will hash on the key of "1, 1", and subsequently increment $S^F_{b_1 b_2}(1, 1)$ by 3.5. The new value of $S^F_{b_1 b_2}(1, 1)$ will thus be 9.8.

Finally, let us assume that the SUM-1 algorithm is asked to recalculate the value of $S_e^F(\bar{x}_1)$. The SUM-1 algorithm will then respond by returning to the user the updated value of 9.8.

The SUM-1 algorithm is obviously a very simple computer program. In the next section we will watch it operate as a subroutine which gives valuable information to the SUM-2 and SUM-3 procedures.

Remark 4.3.C. At the beginning of this section, the E-1 predicates were defined as those predicates which were either conjunctions of several atomic equality predicates, or were the trivial predicate of "TRUE". The predicate of "TRUE" was mostly ignored in this section because it is very easy to process. In the case of the predicate of "TRUE", the SUM-1 algorithm degenerates into the trivial procedure which continually assures that somewhere a record is stored in computer memory which indicates the value of the sum of

all of the F(y) quantities. Obviously, this sum is updated whenever a y-record is either inserted into or deleted from R_y. Also, obviously the value of this sum will be returned to the user everytime he asks for a sum calculation relative to the predicate of "TRUE".

4.4 The SUM-2 and SUM-3 Algorithms

Let $e(x, y)$ denote an E-3 predicate, and $S_e^F(\bar{x})$ denote its corresponding $F(y)$ sum. In this section, it will be proven that each such E-3 predicate will have an associated sequence of E-1 predicates (denoted as $e_1 e_2 \cdots e_k$) and an associated sequence of constants (denoted as $c_1 c_2 \cdots c_k$) such that the following equality will hold for all values of x:

1) $$S_e^F(x) = c_1 S_{e_1}^F(x) + c_2 S_{c_2}^F(x) + \cdots + c_k S_{e_k}^F(x) \quad .$$

The right-hand side of equation 1 will henceforth be called e's E-1 decomposition. The main theorem of this section will thus state that all E-3 predicates will have such E-1 decompositions. This E-1 Decomposition Theorem is significant because it will suggest a very natural and efficient technique of processing E-3 predicates by calling the SUM-1 algorithm as a subroutine.

The Decomposition Theorem will be proven in two stages in this section. It will be first proven for the special case of "E-2 predicates", and subsequently it will be generalized for the class of the E-3 predicates. The definition of E-2 predicates is given below:

Definition 4.4.A. Let $e^*(x, y)$ denote an E-1 predicate. A predicate $e(x, y)$ will be said to be an E-2 predicate if it is either an E-1 predicate or if it is a conjunction of an E-1 expression with several "$x.a \neq y.b$"

terms, similar to equation 2:

2) $$x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND} \cdots x.a_m \neq y.b_m \text{ AND } e^*(x, y) \quad .$$

Example 4.4.B. Several E-2 predicates are shown in equations 4 through 7. Note that equation 7 is an E-2 predicate because Definition 4.4.A indicated that all E-1 predicates are E-2 predicates.

4) $x.a_1 \neq y.b_1$ AND $x.a_2 \neq y.b_2$ AND $x.a_3 = y.b_3$

5) $x.a_1 \neq y.b_1$ AND $\{x.a_2 = y.b_2$ AND $x.a_3 = y.b_3\}$

6) $x.a_1 \neq y.b_1$ AND $x.a_2 = y.b_2$

7) $x.a = y.b$.

Example of some rather unusual E-2 predicates are given in equations 8 and 9. Note that equation 8 is an E-2 predicate because the condition "TRUE" is an E-1 predicate. Equation 9 will also be regarded to be an E-2 predicate because it is obviously logically equivalent to 8

8) $x.a \neq y.b$ AND TRUE

9) $x.a \neq y.b$.

LEMMA 4.4.C. Every E-2 predicate has an E-1 decomposition.

Proof: An E-2 predicate will be said to have a complexity of m if it contains precisely m inequality terms of the form: "$x.a \neq y.b$."

Lemma 4.4.C is easiest to prove if an inductive argument on the size of m is used. The lemma will be first proven for the case where $m = 0$ and subsequently the inductive step will be verified.

Case $m = 0$: This part of the lemma is very easy to prove. Note that the definition of the complexity of an E-2 expression indicates that zero-complexity expressions must contain no "$x.a \neq y.b$" terms. It thus follows that zero-complexity predicates are E-1 expressions, and they are therefore their own 1-1 decomposition. Hence, the lemma has been trivially confirmed for the case of these zero complexity expressions.

Inductive Case: Let us assume that the lemmas has been proven for the case of all predicates of M-1 complexity, and let us further assume that the symbol $e^*(x, y)$ denotes an E-1 expression. The general form of an E-2 expression with complexity m is given below:

10) $\{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND} \cdots \text{AND } x.a_m \neq y.b_m \text{ AND } e^*(x,y)\}$.

Our discussion will center around the above equation 10. We will prove the lemma by demonstrating that this equation has an E-1 decomposition.

In the discussion which follows, $e(x, y)$ will denote the predicate given in the preceding equation 10, and $e_1^*(x, y)$ and $e_2^*(x, y)$ will denote the two predicates defined below:

11) $e_1^*(x,y) = \{x.a_1 \neq y.b_1 \& x.a_2 \neq y.b_2 \& \cdots x.a_{m-1} \neq y.b_{m-1} \& e^*(x,y)\}$

12) $e_2^*(x, y) = \{e_1^*(x, y) \text{ AND } x.a_m = y.b_m\}$.

Note that the definitions of the $e(x, y)$, $e_1^*(x, y)$ and $e_2^*(x, y)$ predicates jointly imply that these three predicates satisfy the equality:

13) $S_e^F(x) = S_{e_1^*}^F(x) - S_{e_2^*}^F$.

This equation will be used to prove the lemma. Essentially, it will be shown that the E-1 decompositions of predicate $e(x, y)$ is produced by taking the E-1 decomposition of $e_1^*(x, y)$ and subtracting from it the E-1 decomposition of $e_2^*(x, y)$.

The formal proof begins with the observation that $e_1^*(x, y)$ and e_2^* both have complexities equal to $m-1$. The inductive hypothesis is thus applicable, and it implies that both of these expressions have E-1 decompositions. Let the terms on the right-hand side of equation 14 and 15 denote the relevant E-1 decompositions of $e_1^*(x, y)$ and $e_2^*(x, y)$:

14) $S_{e_1^*}^F(\bar{x}) = \sum_{i=1}^{K} c_{1,i} \cdot S_{e_{1,i}}^F(\bar{x})$

15) $S_{e_2^*}^F(\bar{x}) = \sum_{i=1}^{K} c_{2,i} \cdot S_{e_{2,i}}^F(\bar{x})$.

Clearly, equations 13 through 15 imply that the E-1 decomposition of $e(x, y)$ is:

16) $$S_c^F(\bar{x}) = \sum_{i=1}^{K} c_{1,i} \cdot S_{e_{1,i}}^F(\bar{x}) - \sum_{i=1}^{K} c_{2,i} S_{e_{2,i}}^F(\bar{x}) \quad .$$

Hence, the lemma is proven since $c(x, y)$ has been shown to have an E-1 decomposition.

Q. E. D.

Example 4. 4. D. Let $c(x, y)$, $e_1(x, y)$ and $e_2(x, y)$ denote three predicates given below:

14) $$c(x, y) = \{x.a_1 \neq y.b_1 \text{ AND } x.a_2 = y.b_2\}$$

15) $$e_1(x, y) = \{x.a_2 = y.b_2\}$$

16) $$e_2(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2\} \quad .$$

Note that e_1 and e_2 are E-1 expressions. The E-1 decomposition of $c(x, y)$ is clearly:

17) $$S_c^F(\bar{x}) = S_{e_1}^F(\bar{x}) - S_{e_2}^F(\bar{x}) \quad .$$

It will frequently be useful in this thesis to speak of a set of L predicates: $e_1(x, y)$, $e_2(x, y) \cdots e_L(x, y)$ which are fully disjoint. This concept is defined below.

Definition 4. 4. E. The predicates $e_1(x, y)$, $e_2(x, y) \cdots e_L(x, y)$ will be said to be fully disjoint if it is logically impossible for any xy-ordered pair to satisfy more than one of those predicates at the same time.

Example 4. 4. F. The predicates $e_1^*(x, y)$ and $e_2^*(x, y)$, given in equations 18 and 19, are clearly fully disjoint since they cannot be simultaneously satisfied:

18) $$e_1^*(x, y) = \{x.a \neq y.b\}$$

19) $$e_2^*(x, y) = \{x.a = y.b\} \quad .$$

LEMMA 4. 4. G. Every E-3 predicate has an E-1 decomposition.

Proof: Let $c(x, y)$ denote an E-3 predicate. Note that every E-3 predicate can be viewed as a disjunction of several fully disjoint E-2 predicate if the standard disjunction normalized form is used. *Such an interpretation of* predicate $c(x, y)$ is shown on the right-hand side of equation 20. In that equation, the symbols $e_1^* e_2^* \cdots e_L^*$ represent a sequence of L fully disjoint E-2 predicates:

20) $$c(x, y) = \{e_1^*(x, y) \text{ OR } e_2^*(x, y) \text{ OR} \cdots \text{OR } e_L^*(x, y)\} \quad .$$

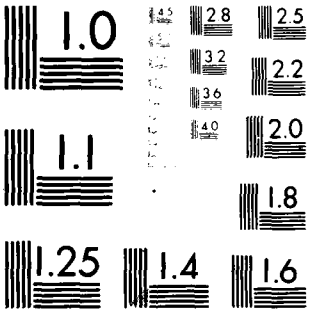
Note that equation 20 and the fact that the e_i^* predicates are disjoint collectively imply equation 21:

21) $$S_c^F(x) = S_{e_1^*}^F(x) + S_{e_2^*}^F(x) + \cdots + S_{e_L^*}^F(x) \quad .$$

The remainder of the proof rests on two simple observations. The first is that Lemma 4. 4. C stated that all E-2 expressions (including the above e_i^* predicates) have E-1 decompositions. The second is that equation 21 implies

AD-A110 139 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 5/2
PREDICATE-ORIENTED DATABASE SEARCH ALGORITHMS.(U)
MAY 78 D E WILLARD N00014-76-C-0914
UNCLASSIFIED TR-20-78 NL

2 of 3
AD
A110139



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

that the sum of these e_i^*-decompositions is equal to the decomposition of the $e(x, y)$ predicate. Hence, the proof is completed since it has been demonstrated that $e(x, y)$ has an E-1 decomposition.

Q. E. D.

Example 4.4.H. Let us calculate the E-1 decomposition for the E-3 predicate defined below:

22) $$e(x, y) = \{x.a_1 \neq y.b_1 \text{ OR } x.a_2 \neq y.b_2\} \quad .$$

The following five predicates will be used in this example:

23) $$e_1^*(x, y) = \{x.a_1 \neq y.b_1\}$$

24) $$e_2^*(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 \neq y.b_2\}$$

25) $$e_3(x, y) = \{x.a_1 = y.b_1\}$$

26) $$e_4(x, y) = TRUE$$

27) $$e_5(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2\} \quad .$$

Note that in this example $e_1^*(x, y)$ and $e_2^*(x, y)$ are E-2 predicates whose E-1 decompositions are:

28) $$S^F_{e_1^*}(x) = S^F_{e_4}(x) - S^F_{e_3}(x)$$

29) $$S^F_{e_2^*}(x) = S^F_{e_3}(x) - S^F_{e_5}(x) \quad .$$

Also note that the example has been constructed so that e_1^* and e_2^* will be

fully disjoint E-2 predicates which satisfy the condition:

30) $$S^F_e(x) = S^F_{e_1^*}(x) + S^F_{e_2^*}(x) \quad .$$

Equations 28 through 30 thus imply that the E-1 decomposition of $e(x, y)$ is

31) $$S^F_e(x) = (S^F_{e_4}(x) - S^F_{e_3}(x)) + (S^F_{e_3}(x) - S^F_{e_5}(x))$$
$$= S^F_{e_4}(x) - S^F_{e_5}(x) \quad .$$

The decomposition theorems that were proven in this section are important because they suggest a very simple and efficient procedure for calculating sums relative to E-2 and E-3 predicates. This procedure will be called the SUM-3 algorithm. It will process E-2 and E-3 expressions by first decomposing them into E-1 parts and then asking the SUM-1 algorithm to perform the remaining work on the derived E-1 components. The rest of this section will formally define the SUM-3 procedure and demonstrate its correctness and $\Theta(1;1)$ runtime efficiency.

This discussion will be divided into three parts, where the compiler, the data-retrieval module, and the data-modification module of the SUM-3 algorithm are separately examined. The first of these three modules will be shown to have an $\Theta(N)$ runtime, and the latter two modules will have $\Theta(1)$ runtimes. Each of these three modules will make several more or less obvious subroutine-calls to the corresponding modules of the SUM-1 algorithm.

The discussion of the SUM-3 algorithm will begin with an examination of its compiler. This compiler will be activated whenever the user supplies the algorithm with an $e(x, y)$ predicate, an $F(y)$ function, and a request for future S_e^F sums to be calculated with maximum efficiency. Let us assume that equation 32 represents the E-1 decomposition associated with the user's specified $e(x, y)$ predicate, and that $H_{e_1}^F$ represents the additive hash corresponding to this equation's $S_{e_1}^F$ term.

32) $$S_e^F(x) = c_1 S_{e_1}^F(x) + c_2 S_{e_2}^F(x) + \cdots + c_L S_{e_L}^F(x) \quad .$$

In this case, the SUM-3 compiler will make L subroutine-calls to the SUM-1 compiler and ask SUM-1 to construct the L distinct additive hashes of: $H_{e_1}^F H_{e_2}^F \cdots H_{e_L}^F$. These additive hashes will collectively constitute the database which the SUM-3 algorithm will manipulate. In the rest of our discussion, we will see how this database is useful.

The SUM-3 data-retrieval module will search the preceeding whenever the user instructs it to calculate a $S_e^F(\bar{x})$ value. A simple two-step procedure is employed for such calculations.

In the first step, the SUM-3 algorithm will make L subroutine-calls to the SUM-1 algorithm and instruct this procedure to use the $H_{e_1}^F, H_{e_2}^F \cdots H_{e_L}^F$ additive hash files for the purposes of calculating the

values of

$$S_{e_1}^F(\bar{x}), S_{e_2}^F(\bar{x}) \cdots S_{e_L}^F(\bar{x}) \quad .$$

In the second step, the data-retrieval module will take these $S_{e_i}^F(\bar{x})$ values and use the arithmetic formula given in equation 32 to combine them and derive the value of $S_e^F(x)$. Note that Theorem 4.4.G stated that all E-3 predicates will have an E-1 decomposition formula, similar to equation 32. It thus follows that the preceeding data-retrieval algorithm will be applicable to all E-3 predicates.

The third part of the SUM-3 algorithm will be the data-modification module. This module will be activated whenever the user indicates that a y-record should be either inserted into or deleted from the database. In either case, the data-modification module's response will be the same, and it will make L subroutine calls to the SUM-1 data-modification module and instruct that procedure to appropriately modify the various $H_{e_1}^F H_{e_2}^F \cdots H_{e_L}^F$ additive hash files. When these changes are completed, the database will have been appropriately updated so that future $S_e^F(\bar{x})$ calculations will reflect the just mentioned insertion or deletion operation.

The discussion of the SUM-3 algorithm will now conclude with a theorem describing its correctness and an example.

THEOREM 4.4.I. The SUM-3 algorithm, which has been described in this section, will

A) Operate correctly;

B) and perform its work in $\Theta(1;1)$ runtime.

Proof of Proposition A: Note that the SUM-3 algorithm was basically built around the two assumptions that:

I) All E-3 predicates can be decomposed into E-1 components.

II) And that these E-1 predicates can be correctly processed by the SUM-1 algorithms.

These two assumptions are verified respectively by Theorems 4.4.G and 4.3.A. An examination of the three modules of the SUM-3 algorithm clearly indicates that these modules were basically straightforward, trivial applications of the previous two theorems. Thus, the correctness of the entire SUM-3 algorithm follows from these two theorems. Q.E.D.

Proof of Proposition B: An examination of the SUM-3 algorithm indicates that each of the three modules of this algorithm will make L subroutine calls to the corresponding lower-level modules of the SUM-1 algorithm. Note that Theorem 4.3.E stated that the compiler of the SUM-1 algorithm will consume $\Theta(N)$ time and that its data-retrieval and data-modification modules will have $\Theta(1)$ run

time. It thus follows that SUM-3's compiler will consume LN time, and its data-retrieval and data-modification modules will each consume L time. Also note that the integer L, in the preceeding estimates, was a constant whose size depended only on the predicate $e(x,y)$. L should therefore be regarded as a runtime coefficient (because its magnitude is unrelated to the size of the database). Runtime coefficients are not included in our estimates of orders of magnitudes. It thus follows that the SUM-3 compiler will consume $\Theta(N)$ runtime, and its data-retrieval and data-modification modules will consume $\Theta(1)$ runtime. An algorithm whose data-retrieval module and data-modification module both consume $\Theta(1)$ will, of course, be said to have $\Theta(1;1)$ runtime. The SUM-3 algorithm is thus such a procedure. Q.E.D.

Example 4.4.J. Let $e(x,y)$, $e_1(x,y)$ and $e_2(x,y)$ be the predicates which were defined in equations 14 through 16 of Example 4.4.D. Note that Example 4.4.D showed that the E-1 decomposition of $e(x,y)$ is:

$$33)' \qquad S_e^F(x) = S_{e_1}^F(x) - S_{e_2}^F(x) \quad .$$

In this example, we will watch the SUM-3 algorithm calculate $S_e^F(\bar{x})$ values for this predicate.

The SUM-3 algorithm will, of course, be initially activated when the user instructs the compiler to develop the capacity for handling predicate $e(x,y)$. At that time, the compiler will calculate the E-1 decomposition of

$e(x, y)$ (which was given in equation 33). Note that this decomposition breaks predicate $e(x, y)$ into e_1 and e_2 components. It thus follows that the SUM-3 algorithm will need the corresponding additive hash files of $H^F_{e_1}$ and $H^F_{e_2}$. The SUM-3 compiler will therefore call the SUM-1 compiler and instruct it to construct these two hash files.

The data-retrieval module will use these files in the obvious manner during its calculation of $S^F_e(x)$ values. The relevant procedure will have two steps. It will begin by calling the SUM-1 algorithm and asking that procedure to determine the values of $S^F_{e_1}(\bar{x})$ and $S^F_{e_2}(\bar{x})$. Subsequently, the SUM-3 algorithm will perform a $S^F_{e_1}(\bar{x}) - S^F_{e_2}(\bar{x})$ subtraction to determine the value of $S^F_e(\bar{x})$.

Obviously the SUM-1 algorithm's calculation of the $S^F_{e_1}(\bar{x})$ and $S^F_{e_2}(\bar{x})$ quantities was crucial to the procedure which the SUM-3 algorithm used in the preceeding paragraph. The details of exactly how SUM-1 performs this calculation will not be provided in this example because the SUM-1 algorithm was adequately discussed in the last section. All that is necessary to understand at this vantage point is that SUM-1 uses the $H^F_{e_1}$ files to calculate the $S^F_{e_1}(\bar{x})$ values.

The third module of the SUM-3 algorithm will be its data-modification module. This module will obviously update the $H^F_{e_1}$ and $H^F_{e_2}$ hash files whenever a y-record is either inserted or deleted in the R_y relation.

Throughout this thesis, the symbol $I_e(\bar{x})$ will be used to denote the number of y-records which satisfy $e(\bar{x}, y)$. Theorem 4.4.1 thus implies the following corollary.

<u>COROLLARY 4.4.K.</u> A COUNT algorithm can be designed which will calculate $I_e(\bar{x})$ values for E-3 expressions in $\Theta(1;1)$ runtime.

<u>Proof:</u> The COUNT algorithm can be regarded as a specialized version of the SUM algorithm. Thus, if $F(y)$ is chosen to be that special function which maps all y-records onto the integer 1 (in the additive group of integers) then clearly $S^F_e(\bar{x})$ will be equal in this case to $I_e(\bar{x})$. Hence, the COUNT algorithm can calculate $I_e(\bar{x})$ values by simply calling the SUM algorithm as a subroutine. Note that Theorem 4.4.1 indicated that such a sum-calculating procedure has an $\Theta(1;1)$ runtime. The corollary is therefore proven, since it has been shown that $I_e(\bar{x})$ values can be calculated in this runtime.

<u>Comment:</u> The COUNT algorithm will play a major role in the next chapter of this thesis. It will provide the FIND algorithm with important information

describing the statistical distribution of the data. The FIND algorithm will use that information to determine which of several equivalent search paths are likely to have the best runtime.

CHAPTER 5

5.1 Chapter Overview

Let the symbol e denote an E-3 predicate, $\bar{x}$ denote an argument of this predicate, $Y_e(\bar{x})$ denote the set of y-records which satisfy $e(\bar{x}, y)$, and $I_e(\bar{x})$ denote the size of this $Y_e(\bar{x})$ set. The purpose of this chapter will be to develop a procedure, called FIND-3, which will construct the $Y_e(\bar{x})$ sets for E-3 expressions in $\Theta(1 + I_e(\bar{x}); 1)$ worst-case hash-runtime. The FIND-3 procedure will be a very important algorithm because it will possess the literally perfect runtime order of magnitude.

The discussion in this chapter will be divided into 4 major sections. The first three sections will discuss FIND-1 algorithms. This procedure will be designed to process the "NEG-1 subclass" of E-3 expressions. Section 5.2 will define NEG-1 expressions and explain the intuitive idea behind this FIND-1 procedure. A more detailed discussion of the FIND-1 algorithm will take place in sections 5.3 and 5.4. Those sections will illustrate how this algorithm respectively performs data-retrieval and data-modification operations.

The last part of this chapter will be section 5.5. That section will define the FIND-2 and FIND-3 algorithms. These procedures will be designed to perform FIND operations on the respective classes of E-2 and E-3 expressions. Obviously the main goal of this chapter will be to prove that the FIND-3 algorithm can operate in $\Theta(1 + I_e(\bar{x}); 1)$ worst-case hash-runtime.

The FIND-1, FIND-2 and FIND-3 procedures will be very natural outgrowths of the last chapter's SUM and COUNT algorithms. The latter two procedures will be used to gather statistical information describing the distribution of the data. The FIND algorithms will use this statistical information to determine which of several equivalent database designs is likely to produce the optimal or near-optimal results. In this chapter, it will be proven that the FIND algorithms can attain $\Theta(1 + I_e(\bar{x}); 1)$ runtime with such techniques.

One final prefatory remark should be made. Sections 5.3 and 5.4 are very complicated. I do not recommend that they be examined on the first reading of this thesis. Indeed, some readers may prefer to omit all of this chapter rather than examine half of it.

5.2 An Intuitive Description of the FIND-1 Algorithm

A predicate $e(x, y)$ will be said to be a NEG-1 expression if it is either the trivial predicate of "TRUE" or a conjuction of several inequality terms similar to equation 1:

(1) $\{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND} \cdots \text{AND } x.a_k \neq y.b_k\}$.

The first goal of this chapter will be to prove that the FIND-1 algorithm can construct its $Y_e(\bar{x})$ sets in $\Theta(1 + I_e(\bar{x}); 1)$ worst-case hash-runtime. The FIND-1 algorithm will be important because the FIND-2 through FIND-8 procedures will make subroutine calls to it.

The FIND-1 algorithm will be one of the two most complicated procedures in this thesis. This section will be intended to explain the intuitive idea behind this algorithm.

The symbol $Q_e(R_y)$ will denote the database that the FIND-1 algorithm will use when it is searching the R_y relation with the $e(x, y)$ predicate. This database will consist of a collection of hashlists. One of these hashlists will be a master-list describing the entire R_y-relation. This hash-list will be denoted as H_{R_y}. The FIND-1 database will usually (not always) contain additional hashlists. These lists will describe subsets of R_y that satisfy conditions similar to

(2) $y.b_1 \neq c_1$ AND $y.b_2 \neq c_2$ AND $\cdots$ $y.b_j \neq c_j$.

Note that there are j terms appearing in equation 2. Often, the $Q_c(R_y)$ database will use sublists generated by such j-termed equations to describe an $c(x, y)$ predicate with K-terms (where $K \geq j$). The next several paragraphs will explain how such sublists can be useful.

These hashlists will be used by the retrieval module of the FIND-1 algorithm whenever the user asks it to construct a $Y_c(x)$ set. A two-part procedure will be used to perform this retrieval task. The first part of this retrieval procedure will be designed to locate one of the smallest hashlists in the $Q_c(R_y)$ database that contains the entire $Y_c(x)$ set. In some cases, the located list will be the H_{R_y} master-file, and in other instances, it will be a sublist. On all occasions, the list-location procedure will find the relevant list in $\Theta(1)$ time.

The second part of the FIND-1 retrieval procedure will make an exhaustive scan through the located hashlist. All the y-records in this list will be tested for whether they satisfy the $c(\bar{x}, y)$ condition. Those y-records that satisfy this condition will be returned to the user (because they constitute his requested $Y_c(\bar{x})$ set).

It should be stated that the above FIND-1 retrieval procedure will constitute only half of the full FIND-1 database algorithm. The other half will

be the FIND-1 data-modification module. This module will be designed to update all the hashlists in the $Q_c(R_y)$ database whenever the user indicates that he desires to have a y-record either inserted into or deleted from the R_y relation.

Note that the database discussed in this section consisted of a H_{R_y} hashlist plus several other special lists describing subsets of R_y. The FIND-1 algorithm will generally be very cautious before constructing the latter type of list. Such lists will be constructed only when the COUNT algorithm (from the last chapter) indicates that the relevant list possesses some unusual statistical characteristic. This cautious approach is necessary because the employment of too many hashlists would be prohibitively expensive.

A more detailed description of the FIND-1 algorithm will be given in sections 5.3 and 5.4. It should be stated that the basic task of this algorithm will be to find some database design that enables its data-retrieval and data-modification modules to operate in the required $\Theta(1 + I_c(x); 1)$ runtime. The FIND-1 algorithm will be shown to use the COUNT algorithm to resolve the traditional trade-off between data-retrieval and data-modification time. Essentially, the FIND-1 algorithm will use the statistical information from the COUNT algorithm to help it determine how this database should be designed. The produced database will be guaranteed to insure that both modules operate in the required runtime. The significant aspect about this

$\Theta(1 + I_e(x); 1)$ runtime of the FIND-1 algorithm is that it will be a worst-case hash estimate.

Finally, it should be once again repeated that sections 5.3 and 5.4 are very complicated. Many readers may prefer to postpone examining these sections until after Chapter 6 is read.

5.3 The FIND-1 Data-Retrieval Module

The FIND-1 data-retrieval module will be discussed in this section, and the data-modification module of this algorithm will be examined in the next section. In both sections, it will be useful to divide general NEG-1 expressions into subclasses according to the number of inequality terms appearing in them. A NEG-1 expression with precisely K such inequality terms will be henceforth called a complexity-K expression. Equation 3 is thus an example of a NEG-1 expression which has a complexity equal to precisely 3:

3) $\{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND } x.a_2 \neq y.b_3\}$.

The FIND-1 algorithm will use a slightly different procedure for processing each separate complexity-class of NEG-1 expressions. Consequently, the symbol "FIND-1-K" will denote the special version of the FIND-1 algorithm which will process complexity-K expressions. For example, this notation suggests that the symbol "FIND-1-3" will denote special version of the FIND-1 algorithm that will process equation 3.

Also, the symbol "NEG-1-K" will be used in this chapter. This symbol will denote the subset of NEG-1 expressions that has a complexity equal to precisely K. Thus, equation 3 is an example of a NEG-1-3 expression.

During this chapter, it will be useful to examine the FIND-1 algorithm from an inductive viewpoint where successively more complicated FIND algorithms are progressively considered. This inductive discussion will follow the standard two-part format which most such arguments use. It will thus begin with an initial description of the FIND-1-Zero algorithm, and it will subsequently use successor arguments to inductively define the FIND-1-K algorithm in terms of the preceeding FIND-1-(K-1) algorithm.

This inductive discussion will begin with the FIND-1-Zero procedure. This algorithm will only be designed to process the predicate of "TRUE" (because "TRUE" is the only NEG-1 predicate whose complexity is equal to precisely zero).

The FIND-1-Zero algorithm will of course be a very simple procedure. Its database will consist of a H_{R_y} hashlist. The FIND-1-Zero algorithm will use this hashlist in the obvious manner. Thus, the entire list will be returned to the user when he asks for the y-records that satisfy "TRUE." Also, the FIND-1-Zero algorithm will continually update this hashlist to reflect the latest records that have been inserted and deleted in R_y.

In some part of this section, it will be useful to refer back to

the preceeding database which the FIND-1-Zero algorithm used. The symbol "$Q_{TRUE}(R_y)$" will be used to denote this database (because this data-structure is designed only for the predicate of "TRUE").

The $Q_{TRUE}(R_y)$ database clearly had a very simple structure. This database and its associated FIND-1-Zero algorithm will be used in the inductive description of the FIND-1-K algorithm. That discussion will be divided into three parts where the FIND-1-K database, FIND-1-K data-retrieval algorithm, and FIND-1-K data-modification algorithm are seperately examined. These three structures will be defined in the standard inductive manner. Thus, the three components of the FIND-1-K algorithm will be defined in terms of the corresponding components of the FIND-1-(K-1) algorithm.

The study of the inductive case will center around the prototype complexity-K NEG-1 expression, given below:

(4) $e(x,y) = \{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND} \cdots x.a_k \neq y.b_k\}$.

In our discussion, the symbol $e_i(x, y)$ will denote that predicate which is identical to $e(x, y)$ except that the "$x.a_i \neq y.b_i$" term has been removed. There will be K such e_i predicates discussed in this section. One such e_i predicate will exist for each possible integer value that i may assume as it varies between 1 and K.

Clearly all the e_i predicates have a complexity equal to K-1 (since

they have one less term than the $e(x, y)$ predicate). The inductive hypothesis is thus applicable to these predicates, and it will allow us to presume that the FIND-1-(K-1) algorithm can efficiently process them. The symbol "Q_{e_i}" will therefore denote the special database which the FIND-1-(K-1) algorithm shall produce when it is processing the e_i predicate.

There will be two other new symbols used in this section. The first symbol will be "$R_{y.\, b \neq c}$". This symbol will denote that subset of the R_y relation which satisfies the "$y.\, b \neq c$" inequality. The second new symbol will be "$Q^{b_i}(c)$." This symbol will be an abbreviation for $Q_{e_i}(R_{y.\, b_i \neq c})$. In other words, $Q^{b_i}(c)$ will denote that database which the FIND-1-(K-1) algorithm uses when it is searching the $R_{y.\, b_i \neq c}$ subrelation with the $e_i(x, y)$ predicate. Note that the inductive hypothesis implies that this database is well defined.

Let us recall that the symbol $Q_e(R_y)$ denotes that database which the FIND-1 algorithm uses when it is searching the R_y relation with the $e(x, y)$ predicate. An explicit description of this database will be given in the next several paragraphs.

Database 4.2.A. There will be two basic data-structures belonging to the $Q_e(R_y)$ database. The first structure will be a hashlist describing the R_y

relation. This hashlist will be once again denoted as H_{R_y}. Note that the description of this thesis's hashing algorithms was given in Section 4.2. Such hashing algorithms will be used to manipulate the H_{R_y} list. The second part of the $Q_e(R_y)$ database will be the "$Q^{b_i}(c)$ structures." These structures were defined two paragraphs earlier to be those databases used by the FIND-1-(K-1) algorithm when it is processing the $R_{y.\, b_i \neq c}$ subrelations. (Each $Q^{b_i}(c)$ structure can be inductively presumed to be a collection of hashlists.)

The $Q_e(R_y)$ database will possess several $Q^{b_i}(c)$ structures (as the values of b_i and c vary). The $Q^{b_i}(c)$ structures will be called "database patches" when they are discussed in the context of the $Q_e(R_y)$ database. The FIND-1-K will attempt to minimize its usage of such patches for reasons related to economizing storage-space and data-modification runtime. Thus, $Q^{b_i}(c)$ patches will be constructed only for specially designated values of b_i and c.

Most of this section will center around the question of which values of b_i and c should be given a constructed $Q^{b_i}(c)$ patch. In that discussion, $Q^{b_i}(c)$ patches will be said to be "fully-constructed," "unconstructed," or "partially-constructed." The meaning of the first two terms is mostly self-evident. The first term refers to a physically constructed data-structure. The

second term likewise refers to an unconstructed data-structure. The third term has a meaning which is somewhat more obscure. It refers to a patch which is currently undergoing construction but which is not yet "fully-constructed."

The FIND-1-K algorithm will use three general rules in determining the construction states of its patches. In the discussion of these rules, K will denote the complexity of the predicate ϵ, N will denote the size of the R_y relation, and $I^b(c)$ will denote the COUNT of the number of y-records that satisfy "$y.b = c$." The following 3 rules will be used to determine the construction states of the patches

i) $Q^b(c)$ will always be fully-constructed when $I^b(c) > N/2K$

ii) $Q^b(c)$ will always be unconstructed when $I^b(c) < N/4K$

iii) $Q^b(c)$ can be in any one of the three construction-states of "fully-constructed," "partially-constructed," or "unconstructed" when the intermediate conditions of $N/4K \leq I^b(c) \leq N/2K$ is satisfied. (In this case, a complex method described later will determine what construction states a patch should be assigned.)

The $Q_c(R_y)$ database can thus be summarized as being a data-structure that consists of an H_{R_y} hashlist and several $Q^b(c)$ patches that satisfy the above three conditions.

Comment: The preceding definition of the $Q_c(R_y)$ database was probably

very confusing. It is possible to give a slightly more intuitive description of this database. Intuitively, each $Q^{b_1}(c)$ patch may be inductively presumed to consist of a series of lists and sublists which describe the $R_{y.b_1 \neq c}$ relation. The $Q_c(R_y)$ database will thus be inductively defined to consist of the H_{R_y} hashlist plus those additional list-structures that are present in the partially and fully-constructed $Q^{b_1}(c)$ patches.

The advantage to the $Q_c(R_y)$ database will be explained in the next two sections. Essentially it will be proven that $\Theta(1 + I_c(x); 1)$ FIND operations are possible in this database. Our discussion will begin with the definition of the FIND-1-K retrieval module.

The definition of this module will be predicated on the inductive assumption that the retrieval module of the FIND-1-(K-1) algorithm was previously defined. We will also inductively assume that this module has an $\Theta(1 + I_c(x))$ retrieval time. Given these assumptions, a FIND-1-K module which has the same properties will be inductively defined.

The FIND-1-K module will make subroutine-calls to the FIND-1-(K-1) modules whenever it needs a $Q^{b_1}(c)$ patch to be searched. The formal description of the FIND-1-K data retrieval algorithm together with relevant theorems and examples are given below:

Algorithm 5.3.R Let $\epsilon(x, y)$ denote the prototype NEG-1 expression given in equation 4. Note that this expression had a complexity equal to K. The following 3-step procedure will construct the user's $Y_c(x)$ sets in

$\Theta(1 + I_e(\bar{x}))$ time:

1) The first step will take $\bar{x}$'s K attributes of $\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_k$, and it will check to see whether any one of these attributes has an associated "fully-constructed" $Q^{b_i}(\bar{x}.a_i)$ patch.

2) The second step will be activated only when such a fully-constructed patch exists. In this case, a subroutine call will be made to the FIND-1-(K-1) algorithm, and the latter procedure will be instructed to search the relevant $Q^{b_i}(\bar{x}.a_i)$ patch with the e_i predicate. The y-records that are produced by that search will be returned to the user because they constitute the $Y_e(\bar{x})$ set which he requested. (In some instances, the element $\bar{x}$ will possess several distinct attributes whose associated $Q^{b_i}(\bar{x}.a_i)$ patches are fully-constructed. On these occasions, a search through any one of these patches will satisfactorily construct the $Y_e(\bar{x})$ set.)

3) The third step will be activated only when the test in step 1 fails (and thus there exists no fully-constructed $Q^{b_i}(\bar{x}.a_i)$ patches). In this case, the FIND-1-K algorithm will make an exhaustive search through the H_{R_y} hashlist, and it will test each y-record for whether it satisfies the $c(\bar{x}, y)$ condition. Those y-records which satisfy this condition will be returned to the

user (since they obviously constitute the requested $Y_e(\bar{x})$ set).

Comment: The intuitive concept behind the previous algorithm can only be understood if the runtime parameters of this algorithm are examined. Note that the above algorithm sometimes used step 2 and other times used step 3 to construct $Y_c(\bar{x})$ sets. This varied search strategy is a reflection of the fact that the competing needs of the FIND-1-K retrieval and modification modules caused $Q^{b_i}(c)$ patches to be fully-constructed only for certain special values of b_i and c. Such a mixed search strategy is necessary to follow if the divergent runtime needs of the two modules are to be simultaneously satisfied. The success of this approach will be proven in Theorems 5.3.D and 5.4.J. Our discussion will now begin with a preliminary Lemma.

LEMMA 5.3.C. Let us recall that the symbols $I^b(c)$ and $I_e(\bar{x})$ denote the COUNTS of the number of y-records that satisfy the respective predicates of $y.b = c$ and $c(\bar{x}, y)$. Also note that N denotes the size of the R_y relation. This R_y relation will satisfy the following three conditions:

i) For fixed attribute b, there will be no more than $4K$ distinct values of constant c which can simultaneously satisfy the inequality: $I^b(c) \cdot N/4K$.

ii) The $Q_c(R_y)$ database will contain no more than $4K^2$ distinct $Q^{b_i}(c)$ patches which are either partially or fully-constructed.

iii) Also, the exhaustive search module (in the third step of the FIND-1-K algorithm) will be invoked only when $I_e(x) \geq N/2$.

Comment: The preceeding lemma is important because it will be used in the proof of the FIND-1 data-modification theorem (which is Proposition 5.3.D). Some readers may prefer to immediately advance to that theorem and save the proof of this lemma for later.

Proof of Lemma 5.3.C. Each of the three propositions in this lemma are fairly simple consequences of the definitions of $I^b(c)$ and $I_e(\bar{x})$.

Let us begin the proof by examining Proposition I. Note that the definition of $I^b(c)$ implies that every collection of distinct c_j constants must satisfy the following inequality (given a N-membered R_y-relation):

$$(5) \qquad I^b(c_1) + I^b(c_2) + \cdots + I^b(c_L) \leq N \quad .$$

Clearly, the above inequality implies that there can never be more than 4K distinct c_j values with $I^b(c_j) > N/4K$. Thus Proposition I is proven, and we now turn our attention to Proposition II.

This proposition is a fairly straightforward consequence of Proposition I and the definition of $Q^{b_i}(c)$ patches. Note that the definition of patches indicated that a $Q^{b_i}(c)$ patch was partially- or fully-constructed only when $I^{b_i}(c) > N/4K$. By applying Proposition I, we thus conclude that each b_i

attribute can possess no more than 4K such partially or fully-constructed $Q^{b_i}(c)$ patches. The remainder of the proof is a consequence of the fact that e is a complexity-K expression. Note that complexity-K expressions are defined to have K distinct b_i attributes and that Proposition I showed that each such b_i-attributes has no more than 4K distinct $Q^{b_i}(c)$ patches. Thus the entire $Q_e(R_y)$ database can possess no more than $4K^2$ such patches, and Proposition II is thereby proven.

Let us now prove Proposition III. We begin by noting that the FIND-1-K algorithm is defined to use exhaustive searches only when none of predicate e's $Q^{b_i}(\bar{x}, a_i)$ patches are fully-constructed. Also note that the definition of $Q_e(R_y)$ databases indicated that such $Q^{b_i}(\bar{x}, a_i)$ patches fail to be fully-constructed only when

$$6) \qquad I^{b_i}(\bar{x}, a_i) \leq N/2K \quad .$$

The previous two sentences imply that this inequality holds when the FIND-1-K algorithm is makeing exhaustive searches.

The remainder of the proof will follow from the definition of the $I^b(c)$ and $I_e(\bar{x})$ COUNTS. Note that the $I_e(\bar{x})$ COUNT for predicate e (given in equation 4) must satisfy the following inequality:

$$I_e(x) \leq N - \sum_{i=1}^{K} I^{b_i}(\bar{x}, a_i) \quad .$$

Clearly, equations 6 and 7 imply that $l_e(\bar{x}) \geq N/2$. Thus the final proposition of the lemma is proven. Q. E. D.

THEOREM 5.3.D. For every value of the integer K, the corresponding FIND-1-K algorithm will

i) Correctly calculate its $Y_e(\bar{x})$ sets;

ii) And it will perform this operation in $\Theta(1 + l_e(\bar{x}))$ runtime. (More specifically, the relevant retrieval algorithm will use approximately $2\, l_e(\bar{x})$ units of runtime after an initial expenditure of $\Theta(1)$ overhead time.)

Proof: The theorem will be proven by induction on the value of K. Note that the case of complexity-zero expressions was discussed earlier in this section. There we saw that the previously cited FIND-1-Zero algorithm would process the predicate of "TRUE" in the manner required by this theorem. The proof of Theorem 5.5.D will therefore be completed if the proposition is also shown to hold for the inductive case.

In this proof, it will be inductively assumed that the theorem holds for the case of the FIND-1-(K-1) algorithm. This inductive assumption will be used to verify that the theorem is also valid for the case of the FIND-1-K algorithm. The proof will be divided into two parts where Propositions I and II are separately confirmed. We now begin the proof with Proposition I.

Proof of Proposition I. Note that the FIND-1-K algorithm used either step 2 or step 3 to construct its $Y_e(\bar{x})$ sets. We will verify the correctness of this procedure by showing that both steps operate correctly.

Let us begin by examining step 2. This step consisted of a subroutine-call to the FIND-1-(K-1) algorithm, where the latter procedure was instructed to search the relevant $Q^{b_1}(\bar{x}, a_1)$ patch with the e_1 predicate. The correctness of this procedure follows mostly from the inductive hypothesis. Note that this hypothesis allows us to presume that the FIND-1-(K-1) algorithm operates correctly. The inductive hypothesis together with the definitions of e_1 and $Q^{b_1}(\bar{x}, a_1)$ jointly imply that the $Y_e(\bar{x})$ sets are properly constructed when step 2 invokes the FIND-1-(K - 1) retrieval algorithm.

It is also easy to show that the $Y_e(\bar{x})$ sets are correctly constructed when step 3 is invoked. This step consisted of an exhaustive scan of the R_y relation. Clearly, all the y-records satisfying $e(\bar{x}, y)$ can be located by such a simple exhaustive scan. Thus, it is apparent that the $Y_e(\bar{x})$ set is properly constructed by this procedure. Hence, Proposition I has been proven since it has been shown that the $Y_e(\bar{x})$ sets are correctly constructed when either step 2 or step 3 is used.

Q. E. D.

Proof of Proposition II. The proof will be divided into three parts where the runtimes of the three steps of the FIND-1-K algorithm are separately examined.

It will be shown that each step consumes no more than $\Theta(1 + I_e(\bar{x}))$ time. The purpose of this discussion is, of course, to verify that the FIND-1-K retrieval algorithm will have an $\Theta(1 + I_c(\bar{x}))$ runtime when it executes all three such steps. The discussion will now begin with the first step of Algorithm 4.2.B.

This step consisted of a testing procedure where the algorithm checked to see whether one of the $\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_k$ attributes possessed a fully-constructed $Q^{b_i}(\bar{x}.a_i)$ patch. The runtime of this step is very easy to determine. Note that Lemma 5.3.C indicated that there will never be more than $4K^2$ distinct $Q^{b_i}(c)$ patches in the $Q_e(R_y)$ database. Clearly, this implies that step 1 can be executed in $4K^2$ time (since in that time, our algorithm can determine whether any of the $4K^2$ patches matches an attribute of $\bar{x}$). Also note that $4K^2$ is a quantity with an $\Theta(1)$ magnitude (because K is a constant whose value is independent of the size of the database). Step 1 will thus have an $\Theta(1)$ runtime, and we will now turn our attention to steps 2 and 3.

The second step is activated only when a relevant $Q^{b_i}(\bar{x}.a_i)$ patches is fully constructed. This step will consist of a subroutine-call where the FIND-1-(K-1) algorithm is instructed to search the just-mentioned patch. Clearly, the inductive hypothesis allows us to presume that the FIND-1-(K-1) algorithm will operate in $\Theta(1 + I_e(\bar{x}))$ runtime. Thus, the second step of the FIND-1-K algorithm also operates in the required runtime, and we now turn our attention to the third step.

The third step of the FIND-1-K algorithm consisted of an exhaustive

scan. This step will clearly require approximately N units of runtime (to scan the N-membered R_y-relation). The efficiency of this scan is a consequence of Lemma 5.3.C. Note that this lemma stated that step 3 is invoked only when $I_c(x) \geq N/2$. Clearly the preceeding statement guarantees that the exhaustive scan procedure will be executed only when N and $I_c(x)$ have the same approximate orders of magnitude. Thus, the third step will consume $\Theta(I_c(x))$ units of time (with the runtime coefficient being no greater than 2). Proposition II has thus been proven (since it has been shown that all three steps of the FIND-1-K data-retrieval algorithm will operate in the required $\Theta(1 + I_e(\bar{x}))$ runtime).

Q.E.D.

Comment: Note that the preceeding Theorem 5.3.D discussed only the data-retrieval aspects of the FIND-1 algorithm. A comprehensive database management system must, of course, perform both data-retrieval and modification modification operations with great efficiency. A description of the FIND-1 data-modification operations will be given in the next section. Some readers may prefer to immediately advance to that section. Other reader may prefer to examine Examples 5.3.E and 5.3.F before proceeding to that section. These examples will help clarify the previous subject of data-retrieval operations.

Example 5.3.E. Let us consider the example of a five-membered R_y relation where $\bar{y}_1$ through $\bar{y}_4$ have y.b values equal to "11" and where

$\bar{y}_5 . b$ = "12." In this example, we will examine the $Q_e(R_y)$ database which is associated with the "x. a ≠ y. b" predicate. Our goal will be to watch the FIND-1 retrieval algorithm interact with this database.

The $Q_{x.a \neq y.b}(R_y)$ database will be governed by the parameters of K, N, and l^b. Note that in this example, K equals 1 (because "x. a ≠ y. b" is a complexity-1 expression); N equals 5 (because 5 elements belong to our R_y relation); and that $l^b(11)$ and $l^b(12)$ equal respectively 4 and 1. Clearly the preceeding implies that

(8) $$l^b(11) > N/2K$$

(9) $$l^b(12) < N/4K \quad .$$

Equations 8 and 9 of course imply that:

i) the $Q^b(11)$ patch is "fully constructed;"

ii) and $Q^b(12)$ is "unconstructed."

In this example, the $Q^b(11)$ patch will be a hashlist containing only the element of y_5 (because only $\bar{y}_5$ satisfies the "y. b ≠ 1" conditions).

The FIND-1 data-retrieval procedure will search the preceeding $Q_{x.a \neq y.b}(R_y)$ database in the manner suggested by Algorithm 5.3.B. Let us consider the example of the two elements, $\bar{x}_1$ and $\bar{x}_2$, defined below

(10) $$x_1 . a = 11$$

(11) $$x_2 . a = 12 \quad .$$

The FIND-1-1 algorithm will take advantage of the $Q^b(11)$ patch when it is producing the $Y_{x.a \neq y.b}(x)$ set for element $\bar{x}_1$. The procedure will thus make r subroutine-call to the FIND-1-Zero algorithm, and it will instruct the latter procedure to find the needed y-records that belong to the $Q^b(11)$ patch. The only element belonging to this patch is $\bar{y}_5$. This element will therefore constitute the $Y_e(x_1)$ set which the user requested.

The FIND-1-1 algorithm will use a very different procedure when it is calculating the $Y_e(\bar{x}_2)$ set. In this case, no constructed $Q^b(12)$ patch is available. The data-retrieval algorithm must therefore rely on an exhaustive scan. It will thus test each of the five elements in R_y for whether they satisfy the "y. b ≠ 12" condition. The elements, $\bar{y}_1$ through $\bar{y}_4$, of course satisfy this condition. They will therefore be returned to the user as the $Y_e(\bar{x}_2)$ set of the "x. a ≠ y. b" predicate.

It must be emphasized that the algorithm in this example is important because of its runtime efficiency. Note that both the $Y_e(\bar{x}_1)$ and $Y_e(\bar{x}_2)$ sets were constructed in approximately $I_e(\bar{x}_1)$ and $I_e(\bar{x}_2)$ amounts of time. It is clearly impossible to substantially improve this runtime since a minimum of $\Theta(I_e(\bar{x}))$ time is required to produce a set of $I_e(\bar{x})$ distinct members. Thus our algorithm had optimal runtime magnitude. In the next example, we will watch the FIND-1 algorithm answer more complicated queries. Again in that example, it will perform its task in optimal runtime magnitudes.

Example 5.3.5. Let us watch the FIND-1 data-retrieval algorithm calculate the $Y_e(\bar{x})$ sets for the $e(\bar{x}, y)$ predicate defined below:

(12) $$e(x, y) = \{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2\} \quad .$$

In this example, we will assume that our R_y relation has 50 elements and that this relation is defined by the following two conditions:

i) Elements $\bar{y}_1$ through $\bar{y}_4$ will have a $y.b_1$-value equal to "11", and elements $\bar{y}_5$ through $\bar{y}_{50}$ will have $y.b_1$ equal to "12".

ii) Elements $\bar{y}_1$ through $\bar{y}_5$ will have a $y.b_2$ value equal to "21", and elements $\bar{y}_6$ through $\bar{y}_{50}$ will have a $y.b_2$ value equal to "22".

The four main defining parameters of the database in this example are K, N, I^{b_1} and I^{b_2}. The value of K equals 2 in the present example (because $e(x, y)$ is a complexity-2 expression). Also, N will clearly be equal to 50 in this example. The main I^b values of this example are defined below:

(13) $$I^{b_1}(11) = 4$$

(14) $$I^{b_1}(12) = 46$$

(15) $$I^{b_2}(21) = 5$$

(16) $$I^{b_2}(22) = 45 \quad .$$

Clearly equations 13 through 16 imply equations 17 through 20

(17) $$I^{b_1}(11) < N/4K$$

(18) $$I^{b_1}(12) > N/2K$$

(19) $$I^{b_2}(21) < N/4K$$

(20) $$I^{b_2}(22) > N/2K \quad .$$

Also, equations 17 through 20 imply that our $Q_e(R_y)$ database will consist of precisely two fully-constructed patches. These patches are the $Q^{b_1}(12)$ and $Q^{b_2}(22)$ patches.

In this example, we will watch the FIND-1-2 algorithms construct the $Y_e(x)$ sets for the elements $\bar{x}_1$ through $\bar{x}_4$ that are defined by the following table:

element-name	$x.a_1$-value	$x.a_2$-value
$\bar{x}_1$	11	22
$\bar{x}_2$	12	22
$\bar{x}_3$	12	21
$\bar{x}_4$	11	21

Let us begin by watching the FIND-1-2 algorithm process the element of $\bar{x}_1$. Note that the second attribute of $\bar{x}_1$ possessed a "fully-constructed" patch (this patch was the $Q^{b_2}(22)$ structure). The FIND-1-2 data-retrieval algorithm will use this patch when it is constructing the $Y_e(\bar{x}_1)$ set. It will

thus make a subroutine-call to the FIND-1-1 algorithm and instruct the latter procedure to search the $Q^{b_2}(22)$ patch with the "$x.a_1 \neq y.b_1$" predicate. The response of the FIND-1-1 algorithm was discussed mostly in the previous example. The database in Example 5.3.E was identical to the present $Q^{b_2}(22)$ patch. In the previous example, we saw that the FIND-1-1 algorithm would answer the user's request with a $Y_e(\bar{x}_1)$ set that contained the singular element of $\bar{y}_5$. The FIND-1-2 algorithm will thus instruct the FIND-1-1 algorithm to return this set to the user.

The FIND-1-2 algorithm will make similar subroutine-calls to the FIND-1-1 algorithm when processing the elements of $\bar{x}_2$ and $\bar{x}_3$. For element $\bar{x}_3$, the FIND-1-1 algorithm will be instructed to search the $Q^{b_1}(12)$ patch. For element $\bar{x}_2$, it may search either the $Q^{b_1}(12)$ or $Q^{b_2}(22)$ patch (it does not matter which of $\bar{x}_2$'s patches are searched because both are fully constructed). The advantage to such subroutine calls to the FIND-1-1 algorithm is of course that they efficiently construct the user's $Y_e(x_2)$ and $Y_e(x_3)$ sets. Both these sets will be constructed in approximately $\Theta(1 + I_e(\bar{x}))$ units of runtime.

The FIND-1-2 algorithm will use a very different strategy for processing element $\bar{x}_4$. Note that neither $Q^{b_1}(11)$ nor $Q^{b_2}(21)$ is fully constructed. Consequently, the FIND-1 algorithm will construct the $Y_e(\bar{x}_4)$ set by making an exhaustive search through the H_{R_y} list in order to

find those y-records that satisfy $e(\bar{x}_4, y)$.

One final remark should be made with reference to this example. Note that each of the elements of $\bar{x}_1$ through $\bar{x}_4$ satisfied the condition that their $Y_e(\bar{x})$ sets were constructed in no more than approximately $2\, I_e(\bar{x})$ time. The significant aspect of all the FIND retrieval algorithms in this thesis is that they will always operate with such efficiency.

The next section of this chapter will complete the discussion of the FIND-1 algorithm by examining its data-modification module.

5.4 The FIND-1 Data-Modification Module

This section will define the procedure that is used by the data-modification module of the FIND-1 algorithm. The discussion in this section will be conducted in an inductive manner similar to that of the previous section. Thus the FIND-1-K version of this procedure will be defined in terms of the preceding FIND-1-(K-1) algorithm. This method of inductive definition is valid because the initial definition of the FIND-1-Zero data-modification procedure was given at the beginning of section 5.3.

The FIND-1-K data-modification algorithm is a very complicated procedure. The algorithm will therefore be approached from two perspectives in this section. This discussion will begin with an intuitive explanation of this algorithm. That topic will be followed by a more formal description. Some readers may prefer to skim the intuitive discussion because the same ideas are repeated in more detail during the formal discussion.

Throughout this section, the reader should bear in mind that the FIND-1-K data-modification procedure is an $\Theta(1)$ algorithm. All the complications in the procedures are due to the constraints imposed by this run-time.

The FIND-1-K data-modification algorithm will be activated whenever the user indicates that a y-record should either be inserted or deleted in the R_y relation. The purpose of the FIND-1-K modification algorithm will be to correctly revise the $Q_c(R_y)$ database in accordance with the user's requests.

A two-part procedure will be used by the FIND-1-K modification algorithm. The first part of this procedure will be a computer-program that updates the various files in the $Q_c(R_y)$ database in accordance with the user's requests. This procedure will be very simple. Note that there are two types of files that appear in the $Q_e(R_y)$ database. These files are the $Q^b(c)$ patches and the H_{R_y} hashlist (the latter notion was defined to be a hashlist describing R_y). Both of these files will be updated in a straightforward manner in accordance with the user's insertion and deletion commands. The update operations in the H_{R_y} list will be performed by the hashing procedures that were described in Section 4.2. The similar update operations in the $Q^b(c)$ patches will be performed by a subroutine-call to the FIND-1-(K-1) algorithm. (Note that the inductive hypothesis allows us to presume that FIND-1-(K-1) algorithm can perform such operations.)

The procedure described in the previous paragraph was clearly quite simple. In the rest of this section, that procedure will be called the update module for the H_{R_y} and $Q^b(c)$ data-structures. This update procedure will obviously constitute a very important part of the FIND-1-K modification algorithm. There will also be a second module used by the FIND-1-K algorithm.

The need for a second module can be understood if we consider the fact that the values of $P^b(c)$ and N will typically vary during the life of our dynamically-changing database. The second module will be designed to alter

the construction-states of the $Q^b(c)$ patches in the manner dictated by the changing values of $I^b(c)$ and N. The general functioning of this module can be understood if we reexamine the patch-construction rules that initially defined our $Q_e(R_y)$ database. These rules indicated how the construction states of the $Q^b(c)$ patches will depend on the values of $I^b(c)$ and N. The patch-construction rules will play a major role in our forthcoming discussion, and they are therefore reproduced below:

i) The $Q^b(c)$ patch will be fully-constructed whenever $I^b(c) > N/2K$.

ii) The $Q^b(c)$ patch will be unconstructed when $I^b(c) < N/4K$.

iii) The $Q^b(c)$ can be in any one of the three construction-states of "fully-constructed," "partially-constructed," or "unconstructed" when $N/4K \leq I^b(c) \leq N/2K$.

The purpose of the second module of the FIND-1-K modification algorithm will thus be to assure that the $Q^b(c)$ patches continue to satisfy i) through iii) as the values of $I^b(c)$ and N vary. This procedure will be henceforth called the patch-construction module.

Most of the remaining discussion in this section will center on the patch-construction module. Our goal will be to develop a highly optimized patch-construction procedure that operates in $\Theta(1)$ worst-case time. It must be emphasized that all the complications in the patch-construction algorithm are

due to the $\mathcal{O}(1)$ runtime constraint. A patch construction procedure would thus be trivial to design if it had an $\mathcal{O}(N)$ instead of an $\mathcal{O}(1)$ runtime constraint.

An example of a patch construction procedure with an (N) worst-case runtime is that procedure which causes

A) the $Q^b(c)$ patch to be fully constructed whenever $I^b(c) > N/2K$;

B) the $Q^b(c)$ patch to be unconstructed in the alternate case where $I^b(c) \leq N/2K$.

The above procedure will be called the "simple" patch-construction procedure. It will not be adequately efficient for our needs. The principal disadvantage to this procedure can be understood if the case is considered where a single insertion operation causes the value of $I^b(c)$ to rise above the N/2K threshold. In this case, the database management system will be forced to spend $\mathcal{O}(N)$ time constructing the new $Q^b(c)$ patch. Such an expenditure of $\mathcal{O}(N)$ time will clearly be prohibitively expensive in most applications. The main goal in the rest of this section will thus be to develop an improved patch-

construction procedure which has an $\Theta(1)$ worst-case runtime.

This improved optimal procedure will be called the "optimal" patch construction procedure. It will be based on the principle of gradually constructing the $Q^b(c)$ patch during the interim period when the value of $I^b(c)$ ranges between N/4K and N/2K. Under such a procedure, all insertion and deletion operations will have an $\Theta(1)$ worst-case runtime (with slightly higher runtime coefficient).

It must be emphasized that the total amount of work constructing patches will be approximately the same under both the optimal and simple procedure. The distinction between these two procedures is that the simple procedure will perform this entire patch-construction task at one given moment in time whereas the optimal procedure will gradually construct the patch during the interim period when $I^b(c)$ ranges between N/4K and N/2K. There are two reasons why the gradual construction approach is preferable. These are that

1) Most users will prefer a predictable procedure that always operates in $\Theta(1)$ time over a less predictable procedure that usually has a lower runtime coefficient but occasionally requires (N) runtime.

2) Also, the simple construction procedure will occasionally be paralyzed by thrashing difficulties. Such problems could occur if the database management system was given a string of 2m commands where the odd numbered commands inserted a record into the database and the even numbered commands deleted a similar record. Under these circumstances, it is possible that the value of $I^b(c)$ could rise and fall in a manner that it exceeds N/2K on every second command. Clearly such conditions are highly undesirable since they would cause every second operation of the simple patch-construction procedure to consume $\Theta(N)$ runtime. Thus the second advantage to the proposed optimal patch construction procedure is that it will avoid such thrashing difficulties.

A formal description of the proposed optimal patch construction procedure will be given in steps 4 through 6 of Algorithm 5.4.G. A more intuitive description of this procedure will be given in the next four paragraphs.

The patch-construction procedure will use two modules to construct $Q^b(c)$ patches. The first module will be a detection procedure that will test the value of $I^b(c)$ to determine whether it exceeds N/4K. The second module will be a patch-producing procedure that will physically construct the required $Q^b(c)$ patches when the preceeding condition is satisfied. Both modules will be executed in a highly optimal manner, and an intuitive description of them is given in the next two paragraphs.

The detection module will rely upon a rotating scan of the H_{R_y} hashlist to perform most of its work. This scan will examine 8K different elements of the H_{R_y} list every time the user instructs the data-modification algorithm to insert or delete a y-record. The H_{R_y} list will be treated as a circular chain in this step. An advancing pointer will thus cause the detection procedure to gradually traverse this circular list. The detection procedure will calculate the $I^b(y.b)$ values of all the y-records it encounters. It will furthermore order that a $Q^b(y.b)$ patch should be physically constructed whenever $I^b(y.b)$ is found to be greater than N/4K.

Such construction operations will be performed by the second module of the patch-construction algorithm. This module will construct $Q^b(c)$ patches by making a gradual walk through the H_{R_y} list. This patch-production module will process 8K distinct elements from the H_{R_y} list each time the user gives an insertion or deletion command. Each of these 8K elements will be tested for whether they satisfy the "$y.b \neq c$" conditions. Those y-records satisfying this condition will be inserted into the $Q^b(c)$ patch.

A formal theorem describing the preceeding patch-construction procedure will be given later in this chapter. The intuitive concept behind this procedure is fairly easy to describe. Note that both the detection and patch-producing modules

of this procedure will make a complete walk of the H_{R_y} hashlist every time the user gives N/8K insertion and deletion data-modification command. Both of these procedures can thus be executed in sequence whenever the user gives N/4K insertion and deletion commands. This N/4K quantity represents the maximum number of data-modification operations transpiring between the time when $I^b(c)$ initially exceeds N/4K and the time when the $Q^b(c)$ patch is fully-constructed. Clearly, the value of $I^b(c)$ cannot change by more than N/4K during this interim period. Thus, the intuitive idea behind this patch-construction procedure is that it will construct the $Q^b(c)$ patches before the required time when $I^b(c) > N/2K$. A more formal description of the FIND-1 algorithm with theorems and examples will be provided in the rest of this section. That discussion will verify the correctness and efficiency of the preceeding concepts.

Algorithm 5.4.G: Let $e(x, y)$ denote a complexity-K NEG-1 expression similar to the prototype expression given below

(21) $$\{x.a_1 \neq y.b_1 \text{ AND } x.a_2 \neq y.b_2 \text{ AND} \cdots x.a_K = y.b_K\} .$$

The FIND-1-K algorithm will use a 6-step procedure to appropriately revise this predicate's $Q_e(R_y)$ database when the user indicates that some y-record should either be inserted or deleted in the R_y relations. In the description of this procedure, the reader should bear in mind that the first 3

steps of this procedure constitute the previously mentioned update module, and the last 3 steps constitute the patch-construction module. The 6 steps that are activated when the user indicates that $\bar{y}$ should be inserted or deleted are described below:

1) The first step will use the element of $\bar{y}$ to update the H_{R_y} hashlist in accordance with the user's command. The hashing algorithm from Section 4.2 will be used to manipulate this list. This hashing algorithm will perform that insertion or deletion operation which is implied by the user's command.

2) The purpose of the second step will be to perform the similar update operation in the $Q^b(c)$ patches. These update operations will be performed with a procedure that executes the following two substeps for each fully and partially-constructed $Q^b(c)$ patch in our database:

2a) The purpose of the first substep can be understood if it is recalled that the $Q^b(c)$ patch only describes the subset of R_y that satisfies the "$y.b \neq c$" condition. This substep will test $\bar{y}$ for whether it satisfies this condition. If $\bar{y}$ satisfies this condition then substep 2b will be instructed to revise the $Q^b(c)$ data-structure in accordance with the user's command.

No change will be made in the $Q^b(c)$ patch when the preceeding inequality fails (because in that case $\bar{y}$ is not a member of the subset described by $Q^b(c)$).

2b) The second substep will be activated only when the previous inequality succeeds. In that case, the relevant insertion or deletion operation will be performed via a subroutine-call to the FiND-1-(K-1) data-modification module. The latter procedure will thus be instructed to take the element $\bar{y}$ and to use it to modify the $Q^b(c)$ patch in the manner suggested by the user's insertion or deletion command. (Note that the procedure in this substep is executed in both the case of insertion and deletion commands and in both the instances of partially and fully-constructed patches. One minor additional substep will be required for the special case of deletion operations in a partially-constructed patch. In that case, our algorithm must check to see whether the given record exists before it attempts to delete that record. An entry for the relevant y-record may not exist in a partially-constructed patch, because of the incomplete nature of its construction. Obviously, substep 2b will delete

a y-record from a "partially-constructed patch" only if that record exists in this patch.)

3) The third step of the data-modification algorithm is an implicit step where the various additive-hash files associated with the $I^b(c)$ COUNT functions are updated. Such update operations will be performed for each of the $I^{b_1}, I^{b_2}, \cdots, I^{b_K}$ COUNT functions. It is necessary to update these functions because steps 4 and 6 of the present algorithm will make subroutine-calls to them. Obviously, all update operations in the COUNT files will be executed using techniques from the last chapter. I have called this step an implicit step because it is always implicitly assumed in this thesis that previously described algorithms are continually updating their files.

4) The fourth step of the data-modification algorithm is the first step that is specifically related to the patch-construction task. Step 4 will be the detection component of that task. The general purpose of step 4 will be to locate those values of b and c whose $I^b(c)$ COUNT is greater than N/4K and to instruct Step 5 to construct the $Q^b(c)$ patches for these values of b and c. The specific procedure in Step 4 will consist of the following 3 substeps:

4a) The first substep will treat the elements in the H_{R_y} list as a "circular chain" where the last list element is followed by the first list element as one moves around the H_{R_y} list in a clockwise direction. This step will make a "rotating-scan" around this list where 8K new list elements are collected each time this step is invoked. A special pointer will also be advanced 8K position through this list so that two consecutive invocations of this step will locate two consecutive sequences of 8K elements.

4b) The second substep will examine the 8K elements that were gathered by substep 4a. For, each such y-element, this step will take the K attributes of $y.b_1, y.b_2 \cdots y.b_K$, and it will calculate the associated $I^{b_i}(y.b_i)$ values. Such calculations of $I^{b_i}(y.b_i)$ will be made via subroutine-calls to the last chapter's COUNT algorithm. This step will also test each $I^{b_i}(y.b_i)$ value for whether it satisfies the "$I^{b_i}(y.b_i) \geq N/4K$" condition. At the end of this step, those $y.b_i$ values that satisfy this

condition will be listed.

4c) The third substep will examine every $y.b_1$ value appearing on the previously mentioned list. Each such y-record will be tested for whether its associated $Q^{b_1}(y.b_1)$ patch is at least partially constructed. This step will order that construction should begin on those tested patches which are presently unconstructed. Such a construction order will be implemented by putting a special "partial-construction-pointer" at the end of the H_{R_y} list and by raising a flag which indicates that this pointer will be used to construct the required $Q^{b_1}(y.b_1)$ patch.

5) The fifth step of the data-modification algorithm will use the previous pointer and flag to construct the various $Q^b(c)$ patches. This step will take each partially-constructed patch which exists in the database, and it will execute the following two-step procedure for that $Q^b(c)$ patch.

5a) The first substep will advance the "partial-patch pointer" through 8K distinct elements in the H_{R_y} list. Each of these 8K elements will be tested

against the "$y.b \neq c$" condition. This substep will instruct the FIND-1-(K-1) algorithm to insert those y-records satisfying this inequality into the relevant $Q^b(c)$ patch.

5b) The second substep will check the location of the "partial patch pointer." Clearly, the entire $Q^b(c)$ patch will have been constructed if that pointer has reached the beginning of this list. In that case, this substep will disband the "partial-patch-pointer" and order that a "fully constructed" flag be raised over the $Q^b(c)$ patch.

6) The purpose of the last step of the FIND-1-K modification algorithm is to transform constructed patches into unconstructed patches when the value of $I^b(c)$ falls below N/4K. This step will operate in the straightforward manner. It will ask the COUNT algorithm to calculate the $I^b(c)$ values for each $Q^b(c)$ patch that exists in our $Q_e(R_y)$ database. The "fully-constructed" and "partially constructed" flags will be dropped for those patches which fail to satisfy the $I^b(c) \geq N/4K$ condition. Also, the memory consumed by these patches will be deallocated and returned to the system-garbage-collector.

The rest of this section will prove the efficiency and correctness of the preceeding algorithm. This discussion will begin with two preliminary lemmas. Some readers may prefer to postpone the proofs of these lemmas until after examining the main theorems of this chapter.

LEMMA 5.4.H. Step 4 of the preceeding FIND-1-K data-modification algorithm will cause the $Q^b(c)$ patch to be either partially or fully-constructed whenever $I^b(c) \geq 3N/8K$.

Proof: Note that there must be a minimum of $N/8K$ insertion and deletion data-modification operations between time when $I^b(c)$ is initially less than $N/4K$ and the final time when it becomes larger than $3N/8K$. Also note that Step 4 of Algorithm 5.4.G will examine $8K$ distinct $I^b(y, b)$ values during each data-modification operation. The entire N-membered H_{R_y} list will thus be traversed during this period of time when $I^b(c)$ varies between $N/4K$ and $3N/8K$. Clearly, the element c must be processed during this time. Step 4b of this algorithm will thus notice that $I^b(c) > N/4K$ during this period. Step 4c will consequently order that the partial-construction flag be raised for the $Q^b(c)$ patch. Also note that Algorithm 5.4.G is incapable of forcing a $Q^b(c)$ patch to regress back to an unconstructed state as long as $I^b(c) > N/4K$. It thus follows that this patch will remain in either the partially or fully-constructed once the partial construction flag is raised. Hence the patch will be either partially or fully-constructed when $I^b(c) > 3N/8K$.

Q. E. D

LEMMA 5.4.I. Let us make the inductive assumption that the FIND-1-(K-1) algorithm operates correctly. Under this assumption, the following four propositions will hold

I) Step 2 of the FIND-1-K algorithm will modify the $Q^b(c)$ patches in the correct manner.

II) It is necessarily true that $N/8K$ invocations of Step 5 of the FIND-1-K algorithm will transform a partially-constructed $Q^b(c)$ patch into a fully-constructed patch.

III) A $Q^b(c)$ patch in the $Q_e(R_y)$ database will always be fully-constructed when $I^b(c) > N/2K$.

IV) Steps 4 through 6 of the FIND-1-K algorithm will cause the $Q^b(c)$ patches to always attain the proper construction-state that was required by the definition of $Q_e(R_y)$ databases (these databases were defined in Part 5.3.A of this section).

Proof: We will separately verify the above 4 propositions.

Proof of Proposition I: The validity of this proposition can be seen immediately upon examining Step 2 of the FIND-1 algorithm. The correctness of that step is a trivial consequence of the definition of the $Q^b(c)$ patches and the inductive assumption (which indicates that the FIND-1-(K-1) algorithm properly manipulates these patches).

Q. E. D.

Proof of Proposition II: Note that the procedure in Step 5 examined 8K elements of the H_{R_y} list each time it was invoked. Once again the definition of $Q^b(c)$ together with the inductive hypothesis allow us to presume that this step will insert that subset of these 8K elements into $Q^b(c)$ which belongs in this patch. Thus all N elements of the R_y relation will have been processed after N/8K invocations of this step. At that time, the $Q^b(c)$ patch will have become fully-constructed. Proposition II is therefore proven since we have shown that full-construction is realized after N/8K invocation of Step 5. Q. E. D.

Proof of Proposition III: The proof of this proposition will rest mostly on the observation that there clearly must be at least N/8K data-modification operations between the time when $l^b(c)$ is less than 3N/8K and the later time when it is greater than N/2K. Note that the $Q^b(c)$ patch must be at least partially-constructed during this interim period (by Lemma 5.4.H). It thus follows that Proposition II is applicable to this partially-constructed patch. This proposition will imply that $Q^b(c)$ shall become fully-constructed before $l^b(c)$ is greater than N/2K.

Q. E. D.

Proof of Proposition IV: Note that the preceeding Proposition III indicated that $Q^b(c)$ is fully constructed whenever $l^b(c) > N/2K$. It is also trivial to verify that Step 6 of Algorithm 5.4.G will cause $Q^b(c)$ to

become unconstructed whenever $l^b(c) < N/4K$. Clearly, these combined results will imply that the construction-states of the $Q^b(c)$ patch always satisfy the requirement of the $Q_e(R_y)$ database. Hence, the last proposition is also proven.

Q. E. D.

THEOREM 5.4.J. The FIND-1 modification procedure (described in Algorithm 5.4.G) will

I) Correctly update the $Q_c(R_y)$ database in accordance with the user's insertion and deletion commands;

II) And will perform this task in $\Theta(1)$ worst-case hash-runtime.

Proof: The theorem will be proven by induction on the complexity of e. The FIND-1-Zero version of the FIND-1 algorithm will not be examined in this proof because that trivial procedure was adequately discussed at the beginning of section 5.3. This proof will therefore focus on the more important inductive case.

During this proof, it will be inductively presumed that the FIND-1-(K-1) algorithm satisfies the conditions of the theorem. Given that assumption, it will be shown that the FIND-1-K algorithm also satisfies Propositions I and II. These two propositions will be separately verified. The proof will begin with the verification of Proposition I.

Proof of Proposition I. Note that the purpose of the FIND-1-K algorithm was to update all parts of the $Q_c(R_y)$ database whenever the user

gives an insertion or deletion command. There are four parts of the $Q_e(R_y)$ database which will require updating. The first three parts of this task involve updating the respective H_{R_y} hashlist, $Q^b(c)$ patches, and I^b COUNTING files. The last part of this task will involve adjusting the construction-state of the $Q^b(c)$ patches in accordance with the user's commands. It will be necessary to verify that each of these four tasks are correctly executed to verify Proposition I. This four part analysis is given below:

1) The first part of the FIND-1-K algorithm involves updating the H_{R_y} hashlist in accordance with the user's insertion and deletion commands. This task is performed by Step 1 of Algorithm 5.2.G Note that the procedure in Step 1 updated the H_{R_y} list with the aid of the straightforward hash-insertion and hash-deletion operations. The correctness of this procedure follows from Corollary 4.2.D (which confirmed the correctness of the special hash algorithms used in this thesis).

2) The second part of the FIND-1-K algorithm is designed to update the $Q^b(c)$ patches in accordance with the user's requests. This task is performed by Step 2 of Algorithm 5.4.G. The correctness of that procedure is verified by Proposition I of

Lemma 5.4.I (which stated that the inductive hypothesis allows us to presume that Step 2 operates correctly).

3) The third part of the FIND-1-K algorithm involves updating the I^b-COUNTING files in accordance with the user's insertion and deletion commands. Step 3 of Algorithm 5.4.G is designed to perform this task. The procedure in this step rested on several subroutine-calls to the last chapter's COUNT algorithms. Corollary 4.4.K allows us to presume that the COUNT algorithm will properly perform the needed update operations.

4) The fourth part of the FIND-1-K algorithm involves adjusting the construction-states of the $Q^b(c)$ patches in accordance with the user's insertion and deletion commands. Steps 4 through 6 of Algorithm 5.4.G are designed to perform this task. Proposition IV of Lemma 5.4.I can be used to verify the correctness of these steps (that proposition stated that the inductive hypothesis allowed us to presume that Steps 4 through 6 will operate properly). The correctness of the FIND-1-K algorithms has thus been proven since this procedure was shown to correctly perform all four of its needed tasks.

Q.E.D.

Proof of Proposition II: This proof will be divided into two parts. The first part will separately calculate the runtime of each of the six steps of Algorithm 5.4.G. The second part will add together these runtimes. The total will be shown to be approximately $32K^3 + 20K^2 + 9K + 1$ worst-case time. This quantity will be considered to have an $\Theta(1)$ magnitude (because K is a fixed constant whose value is unrelated to the size of the database).

Our proof will now begin with a separate runtime analysis of each of the six steps of the FIND-1-K algorithm. This six-part analysis is given below:

1) The first step of Algorithm 5.4.G can be best analyzed if Corollary 4.2.D is considered. Note that this corollary indicated that the hash algorithms will always operate in $\Theta(1)$ expected time (including the worst-case where the H_{R_y} hash file is undergoing changes in size). This corollary implies that Step 1 will consume the same $\Theta(1)$ time.

2) The second step of the FIND-1-K algorithm relied upon the FIND-1-(K-1) algorithm to update the various $Q^b(c)$ patches. Note that the inductive hypothesis allows us to presume that each invocation of the FIND-1-(K-1) algorithm will consume $\Theta(1)$ time. Also note that Lemma 5.3.C indicated that there will never be more than $4K^2$ such patches belonging to our

$Q_e(R_y)$ database. Hence, there will never be more than $4K^2$ invocations of the FIND-1-(K-1) algorithm. The total time consumed by this step will thus have an $\Theta(4K^2)$ magnitude in the worst-case. (Usually, this step will require far less runtime because there will usually be less than $4K^2$ patches belonging to the $Q_e(R_y)$ database.)

3) The third step of Algorithm 5.4.G consisted of K invocations of the COUNT algorithm. Each such invocation shall consume $\Theta(1)$ time (by Corollary 4.4.K). Hence Step 3 will consume $\Theta(K)$ time.

4) The fourth step of Algorithm 5.4.G consisted of three substeps. The first substep will take $\Theta(8K)$ time, the second step will take $\Theta(8K^2)$ time, and the third step will consume $\Theta(4K^2)$ time (in the highly unlikely worst-case). Total runtime of Step 4 will therefore have an $\Theta(12K^2 + 8K)$ worst-case magnitude.

5) Step 5 of Algorithm 5.4.G is designed to continue the construction of the "partially-constructed" patches. This step will use the FIND-1-(K-1) algorithm to add $8K$ elements to each patch. The inductive hypothesis will allow us to presume that this step devotes $\Theta(8K)$ time to each partially-constructed

patch. Note that Lemma 5.3.C indicated that the $Q_e(R_y)$ database will never contain more than $4K^2$ such partially-constructed patches. This step will thus consume $\Theta(32K^3)$ time in the very worst-case. (Usually, this step will consume far less time because $Q_e(R_y)$ databases will rarely contain more than two or three partially-constructed patches.)

6) The sixth step will make one subroutine-call to the COUNT algorithm for each partially and fully-constructed patch that belongs to our database. Corollary 4.4.K and Lemma 5.3.C can be used to calculate the runtime consumed by this step. Note that the corollary indicated that each invocation of the COUNT algorithm will consume $\Theta(1)$ time, and that the lemma indicated that no more than $4K^2$ patches will belong to the database. Hence, Step 4 will consume $\Theta(4K^2)$ worst-case time (during its $4K^2$ invocations of this algorithm).

Clearly, the preceeding six points indicated that the total runtime consumed by all six steps of the COUNT-1-K algorithm had an $32K^3 + 20K^2 + 9K + 1$ worst-case magnitude. Note that K in the preceeding term is a constant whose value equals the number of inequality terms in our initial $e(x, y)$ predicate. Hence, the "$32K^3 + 20K^2 + 9K + 1$" quantity can be regarded as a runtime coefficient since the value of this expression is independent of the

size of the database. Algorithm 5.4.G is thus an $\Theta(1)$ algorithm with a moderately large runtime coefficient. The assertion of Proposition II has thus been proven. Q.E.D.

Remark 5.4.K: There are several comments that should be made about the preceeding runtime coefficient. First, it should be stated that the preceeding analysis was based on a worst-case perspective where everything was assumed to go wrong, and it all happened to START GOING WRONG at the SAME MOMENT in time. The runtime coefficient will be far better in the more typical computer applications. For example, the expected coefficient can be RIGOROUSLY PROVEN to be proportional to K for virtually all databases whose records are produced by most standard random-number generators. Secondly, it should be stated that there are many available techniques which will improve this runtime coefficient. Some such techniques will be discussed in Remark 5.4.N. Mostly, runtime-coefficient optimization techniques are ignored in this chapter and thesis because the subject matter is sufficiently complicated without them. Finally, it should be stated that the question of the runtime coefficient is largely academic. The reason for this will become apparent in our discussion of the FIND-2 through FIND-8 algorithms. We will see that these algorithms will generally make subroutine-calls to the FIND-1 algorithm only when K equals one, two, or zero. Thus, the runtime coefficient of the FIND-1 algorithm is adequately small

in most of the intended applications.

Example 5.4.1.: In this example we will watch Algorithm 5.4.G manipulate the $Q_e(R_y)$ database for the predicate given below:

(22) $$e(x, y) = \{x.a \neq y.b\} \quad .$$

The R_y relation will be assumed to initially have 65 elements in this example. These elements will be denoted as $\bar{y}_1 \bar{y}_2 \cdots \bar{y}_{65}$, and their $\bar{y}_i.b$ attributes will be defined to be equal to 1.

During this example, we will watch Algorithm 5.4.G adjust the $Q_e(R_y)$ database as the user gives commands indicating that 135 new elements should be inserted into the R_y relations. These new records will be denoted as $\bar{y}^*_{66}, \bar{y}^*_{67} \cdots \bar{y}^*_{200}$. It will be assumed that these 135 records satisfy the condition $\bar{y}^*_i.b = 25$. The purpose of this example is to illustrate how the FIND-1 will adjust the $Q_e(R_y)$ database in response to the user's command.

The initial state of the $Q_e(R_y)$ database is very easy to compute. The three main parameters of this database are K, N, and the I^b COUNT function. Clearly, K equals 1 (because e contains 1 inequality term), N equals 65 (because our database initially contains 65 y-records), and $I^b(c) = 1$ (for all integer values of c between 1 and 65). The preceeding clearly implies that $I^b(c) < N/4K$ (for all values of c). It thus follows that our $Q_e(R_y)$ database will initially contain no constructed $Q^b(c)$ patches.

In this example, the value of $I^b(25)$ will grow as the $\bar{y}^*$ elements

are inserted into the database. The FIND-1 algorithm will be forced to respond to this fact by eventually constructing the corresponding $Q^b(25)$ patch. Our goal will be to watch how this happens.

In this example, we will assume that user begins by instructing the FIND-1 algorithm to insert y^*_{66} into the $Q_e(R_y)$ database. The FIND-1 algorithm will respond to the user's commands by executing steps 1, 3 and 4. The action of these three steps is described below:

i) Step 1 will insert y^*_{66} into H_{R_y}.

ii) Step 3 will increase $I^b(25)$ from 1 to 2 (because there are now two y-records belonging to R_y with $y.b = 25$).

iii) Step 4 will check the values of $I^b(\bar{y}_1.b)$ through $I^b(\bar{y}_8.b)$. Note that all eight of these integers satisfy the condition: $I^b(c) < N/4K$. Consequently, all eight associated $Q^b(c)$ patches will remain unconstructed.

Note that the preceeding FIND-1 algorithm did not execute Steps 2, 5 or 6. The FIND-1 algorithm will avoid executing these steps when the $Q_e(R_y)$ database contains no partially or fully-constructed patch.

The FIND-1 algorithm will use a similar procedure for inserting $\bar{y}^*_{67}$ into the $Q_e(R_y)$ database. Thus, the H_{R_y} list will be modified, $I^b(25)$ will be updated, and the COUNT-values of $\bar{y}_9$ through $\bar{y}_{16}$ will be checked.

All aspects of this procedure are similar to the insertion of $\bar{y}_{66}$. The details have thus been omitted.

The FIND-1 algorithms will perform similar insertion operation for records $\bar{y}^*_{68}$ through $\bar{y}^*_{88}$. Its response will be very different when record, $\bar{y}^*_{89}$, is encountered. At that time, Step 4 of the algorithm will calculate the COUNT values of records $\bar{y}_{25}$ through $\bar{y}_{32}$. An examination of the $l^b(\bar{y}_{25}.\ b)$ COUNT will reveal that this value has grown to the point that it is now greater than N/4K. Consequently, a "partial-construction flag" will be raised over the corresponding $Q^b(25)$ patch.

This partial-construction flag will cause the $Q^b(25)$ patch to be constructed during the periods when records $\bar{y}^*_{88}$ through $\bar{y}^*_{98}$ are inserted into the database. Step 5 will be executed during each of these insertion operations. Each invocation of this step will process 8 elements on the H_{R_y} list. This step will insert those y-records into $Q^b(25)$ which satisfy the "$y.\ b \neq 25$" inequality.

The full $Q^b(25)$ patch will be constructed at the time when the algorithm is done processing record $\bar{y}^*_{98}$. At that time, the entire H_{R_y} list will have been walked. <u>Note that the $Q^b(25)$ will have realized full construction before the time when $l^b(25)$ has grown to be larger than N/2K.</u> ($l^b(25)$ will attain this larger value only after $\bar{y}^*_{128}$ is inserted.) Thus, our example has shown that the FIND-1 will operate in the manner predicted by the previous

theorems. □

THEOREM 5.4.M. The FIND-1 algorithm is an efficient procedure which will construct the $Y_e(\bar{x})$ sets for B-1 expressions in $\Theta(1 + l_e(x); 1)$ worst-case time. Furthermore, retrieval operations can be performed in $2\, l_e(\bar{x})$ units of time after an initial expenditure of $\Theta(1)$ overhead time.

<u>Proof:</u> This theorem is an immediate consequence of Theorems 5.3.D and 5.4.J. Note that these two propositions showed that the retrieval and modification modules satisfied their respective required conditions. The combined implication of these propositions is that FIND-1 is an $\Theta(1 + l_e(\bar{x}); 1)$ database algorithm. Q. E. D.

<u>Remark 5.4.N.</u> There are many further modules that could be added to the FIND-1 algorithm for purposes of improving its runtime coefficient. These modules were not included in Algorithm 5.3.B or 5.4.G because they would have complicated the proofs. A partial list of useful runtime coefficient strategies is given below. This list does not constitute a complete catalog of all the possible coefficient strategies. One extremely important coefficient-optimization technique was omitted because it was too difficult to summarize. Four useful strategies are mentioned below:

1) The first technique involves eliminating redundancies from the $Q_e(R_y)$ data-structure. In some cases, Algorithm 5.4.G will

produce several identical lists describing the same subset of R_y. A fairly simple module can be added to the FIND-1 algorithm for the purposes of removing these redundancies. A detailed description of that module was not included in this thesis because it would have required new notation and more complicated proofs.

2) A second improvement in the FIND-1 algorithm is possible if a distinction is made between times when the computer is relatively busy and times when it is mostly idle. The task of constructing new patches should be usually shifted to times when the computer would be otherwise idle. Obviously, high-priority computer-time will be conserved if such a method is followed.

3) A further improvement in the FIND-1 procedure is possible if Steps 4 and 6 of Algorithm 5.4.G are revised. These two steps were made very simple for the purposes of theorem-proving. Several straightforward modifications of these steps will improve their runtime coefficient.

4) The runtime coefficient of the FIND-1 algorithm can also be improved if that procedure is fed several parameters describing the likely usage and cost of computer memory and of insertion, deletion, and retrieval operations. It is possible to develop a

FIND-1 algorithm which will use these parameters to minimize the total expected computing cost.

A more detailed description of the above techniques has been avoided in this thesis for two reasons. The first is that most commercial user requests are very simple, and they frequently do not require coefficient optimization. The second and main reason a discussion of coefficient optimization strategies was omitted is that it would have been too lengthy.

Concluding Remark: The usefulness of the FIND-1 procedure will become more apparent in the later sections of this thesis. There we will see how the FIND-1 algorithm constitutes a vital subroutine that is called by the FIND-2 through FIND-8 algorithms.

5.5 The FIND-2 and FIND-3 Algorithms

In this section, we will study the E-2 and E-3 expressions and prove that the FIND-2 and FIND-3 procedures can construct their $Y_e(x)$ sets in $\Theta(1 + I_e(x); 1)$ runtime. The discussion in this section will be divided into two parts which will separately examine the FIND-2 and FIND-3 procedures. Both these algorithms will make fairly straightforward subroutine-calls to the FIND-1 algorithm. The FIND-1 algorithm will thus be very centrally related to the subject matter of this section.

Our discussion will now begin with the E-2 expressions and their FIND-2 algorithm. One possible definition of the E-2 expressions was given in Section 4.4 of this thesis. There are also several other logically equivalent definitions that could have been given to the E-2 class. Let $e^*(x, y)$ denote some NEG-1 expression. In this section, $e(x, y)$ will be regarded to be an E-2 expression if and only if it is either a NEG-1 expression or if it can be written in the following form:

(1) $e(x,y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 = y.b_2 \text{ AND} \cdots x.a_j = y.b_j \text{ AND } e^*(x,y)\}$.

Definition 5.5.A. Let $e(x, y)$ denote an E-2 expression similar to equation 1. Henceforth, $e^*(x, y)$ will be called the NEG-1 component of this expression. Also, $x.a_1, x.a_2 \cdots x.a_j$ and $y.b_1, y.b_2 \cdots y.b_j$ will be called the respective x and y-equality attributes of that expression.

Notation: Let us recall that the symbol $Q_e(R_y)$ denotes that database which the FIND-1 algorithm uses when it is searching the R_y relation with the $e(x, y)$ predicate. Let $R_{y.b_1 = c_1 \text{ AND } y.b_2 = c_2 \text{ AND} \cdots y.b_j = c_j}$ denote that subset of R_y which satisfies the "$y.b_1 = c_1$ AND $y.b_2 = c_2$ AND $y.b_j = c_j$" condition. In this section, symbols such as: $Q_{e^*}(c_1 c_2 \cdots c_j)$ will be frequently used. These symbols will be an abbreviation for $Q_{e^*}(R_{y.b_1 = c_1 \text{ AND } y.b_2 = c_2 \text{ AND } y.b_j = c_j})$. In other words, $Q_{e^*}(c_1 c_2 \cdots c_j)$ will denote that database which the FIND-1 algorithm produces when it is searching the $R_{y.b_1 = c_1 \text{ AND } y.b_2 = c_2 \text{ AND} \cdots y.b_j = c_j}$ relation with the $e^*(x, y)$ predicate.

Database 5.5.B. Let $e(x, y)$ denote an E-2 expression similar to the prototype in equation 1 (whose y-equality attributes were $b_1 b_2 \cdots b_j$ and whose NEG-1 component is $e^*(x, y)$). The symbol $Q\text{-}2(e, R_y)$ will denote that database which the FIND-2 algorithm uses when it is searching the R_y relation with the $e(x, y)$ predicate. This database will have two parts. The first part of the $Q\text{-}2(e, R_y)$ database will be a collection of FIND-1 databases. The previously mentioned symbol of $Q_{e^*}(c_1 c_2 \cdots c_j)$ will denote a typical FIND-1 database in this collection. The second part of the $Q\text{-}2(e, R_y)$ database will be a hash pointer-structure. This hash-structure

will be denoted as $H_{b_1 b_2 \cdots b_j}$. This hash-function will map "$c_1 c_2 \cdots c_j$" tuples onto pointers which are denoted as $H_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$. Each $H_{b_1 b_2 \cdots b_j}(c_1 c_2 \cdots c_j)$ pointer will contain the address of a corresponding $Q_{e^*}(c_1 c_2 \cdots c_j)$ data-structure. The $Q\text{-}2(e, R_y)$ database will also be required to satisfy the further condition that this database will contain the $H_{b_1 b_2 \cdots b_j}$ pointer and Q_{e^*} structure for the "$c_1 c_2 \cdots c_j$" tuple if and only if $R_{y.b_1} = c_1$ AND $y.b_2 = c_2$ AND $\cdots$ $y.b_j = c_j$ is a nonempty set. (This condition means that our database will contain information about "$c_1 c_2 \cdots c_j$" if and only if the corresponding $Q_{e^*}(c_1 c_2 \cdots c_j)$ data-structure describes a nonempty subset of R_y.)

Algorithm 5.5.C. Let $e(x, y)$ denote an E-2 expression. The FIND-2 algorithm will be designed to perform the various database tasks which are associated with searching the $Q\text{-}2(e, R_y)$ database and producing the user's requested $Y_e(\bar{x})$ sets. The FIND-2 algorithm will use the three more or less obvious modules for performing the three corresponding database tasks of retrieval, insertion, and deletion. A description of these three modules is given below:

1) The Retrieval Module will use a two part procedure for constructing the user's $Y_e(x)$ sets. This algorithm will begin with a hash-search for that record indicated by the key of "$\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_j$." If this search finds no $H_{b_1 b_2 \cdots b_j}(\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_j)$ pointer then the FIND-2 algorithm will inform the user that "$Y_e(\bar{x})$ is an empty set." If such a pointer is found, then the FIND-2 algorithm will make a subroutine call to the FIND-1 algorithm. The latter procedure will be instructed to construct the $Y_e(\bar{x})$ set by searching the $Q_{e^*}(\bar{x}.a_1, \bar{x}.a_2 \cdots \bar{x}.a_j)$ substructure with the $e^*(\bar{x}, y)$ predicate.

2) The Insertion Module will be activated when the user instructs the FIND-2 algorithm to insert a y-record into the $Q\text{-}2(e, R_y)$ database. The insertion module will be in many respects very similar to the retrieval module. The insertion module will also use a two-part procedure. It will begin with a hash-search where the algorithm seeks the $H_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ pointer. The subsequent response of the FIND-2 algorithm will depend on whether or not such a $H_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ pointer is found. If this

pointer is found then the FIND-2 algorithm will make a subroutine-call to FIND-1 and instruct it to insert $\bar{y}$ into the $Q_{e^*}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ datasubbase. If no such $H_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ pointer is found in the hashfile then the FIND-2 algorithm will execute the user's command by inserting this pointer and its corresponding one-element $Q_{e^*}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ database into the $Q\text{-}2(e, R_y)$ data-structure.

3) The Deletion Module will be activated when the user instructs the FIND-2 algorithm to delete a y-record. This module will be the more or less obvious inverse of the insertion module. A two step procedure will be used. The first step will hash on "$\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j$" and instruct FIND-1 to remove $\bar{y}$ from the $Q_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2, \cdots \bar{y}.b_j)$ data-structure. The second part of this procedure will consist of a subroutine-call to the COUNT algorithm. The latter procedure will be asked to determine how many records belong to the $Q_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ substructure. If there are no elements left in this structure then the FIND-2 algorithm will deallocate its storage space and disband the corresponding

$H_{b_1 b_2 \cdots b_j}(\bar{y}.b_1, \bar{y}.b_2 \cdots \bar{y}.b_j)$ pointer.

Remark 5.5.D. Note that all three modules of the FIND-2 algorithm made subroutine-calls to hashing algorithms. Such hashing operations should be performed by algorithms similar to those outlined in Section 4.2. The special hashing algorithms of that section will insure that $\Theta(1)$ expected search-time will persist even in the difficult case where the hashfile is undergoing a radical change in size.

THEOREM 5.5.E. The preceeding FIND-2 Algorithm will:

I) Correctly construct the $Y_e(\bar{x})$ sets of E-2 expressions.

II) And perform this database task in $\Theta(1 + l_e(x); 1)$ runtime. (The retrieval module of this algorithm will have an especially efficient runtime. $Y_e(\bar{x})$ sets will thus be produced in $2\, l_e(x)$ time after an expenditure of $\Theta(1)$ overhead time.)

Proof: Note that the FIND-2 algorithm relied on subroutine-calls to the Section 4.2's hashing algorithms, to the COUNT algorithms and to the FIND-1 algorithm. The correctness and efficiency of the three preceeding procedures is verified by respectively Theorems 4.2.D, 4.4.K and 5.4.M. Note that the FIND-2 algorithm was a very simple procedure which applied the previous three algorithms in the obvious manner. Thus, the efficiency and

correctness of the FIND-2 algorithm is a trivial consequence of the previous theorems. Q. E. D.

Example 5.5.F. Let us watch the FIND-2 algorithm manipulate the predicate defined below:

(2) $$e(x, y) = \{x.a_1 \neq y.b_1 \text{ AND } x.a_2 = y.b_2\} \quad .$$

In this example, we will use the preceeding predicate to search the R_y relation that is defined by the following table

element-name	$y.b_1$ attribute	$y.b_2$ attribute
$\overline{y_1}$	11	21
$\overline{y_2}$	11	21
$\overline{y_3}$	11	21
$\overline{y_4}$	11	21
$\overline{y_5}$	12	21
$\overline{y_6}$	11	22
$\overline{y_7}$	12	22

It is useful to begin this example by calculating the y-equality attribute and NEG-1 component of equation 2. Note that its y-equality attribute is obviously "b_2" and that its NEG-1 component is:

(3) $$e^*(x, y) = \{x.a_1 \neq y.b_1\} \quad .$$

There will be two Q_{e^*} datastructures in this example. These will be the $Q_{e^*}(21)$ and $Q_{e^*}(22)$ structures. These two structures will describe the respective subsets of R_y whose $y.b_2$ attribute equal respectively 21 and 22. (In other words, $\overline{y}_1$ through $\overline{y}_5$ will belong to $Q_{e^*}(21)$; and $\overline{y}_6$ through $\overline{y}_7$ will belong to $Q_{e^*}(22)$).

The H_{b_2} hashfile will contain two records in this example. These two records will be denoted as $H_{b_2}(21)$ and $H_{b_2}(22)$. Note that $H_{b_2}(21)$ will be a pointer to $Q_{e^*}(21)$, and that $H_{b_2}(22)$ will be a pointer to $Q_{e^*}(22)$.

In this example, we will watch the FIND-2 algorithm calculate the $Y_e(\overline{x})$ sets for the four x-elements defined below:

element-name	$x.a_1$ attribute	$x.a_2$ attribute
$\overline{x}_1$	11	21
$\overline{x}_2$	12	21
$\overline{x}_3$	12	22
$\overline{x}_4$	12	23

Let us begin by watching the FIND-2 algorithm construct the $Y_e(\overline{x}_1)$ set. In this case, it will hash on the key of "$\overline{x}_1.a_2$." The $H_{b_2}(21)$ pointer will be located by this hash. That pointer contains the address of the $Q_{e^*}(21)$

substructure. The FIND-2 algorithm will therefore instruct the FIND-1 algorithm to search this structure with the e^* predicate and the argument of $\bar{x}_1$. The response of the FIND-1 algorithm was discussed in Example 5.3.E. There we saw that FIND-1 will produce a set that contained the singular element of $\bar{y}_5$. This set will consequently constitute the user's requested $Y_e(\bar{x}_1)$ set.

The FIND-2 algorithm will use a similar procedure when constructing the $Y_e(\bar{x}_2)$ and $Y_e(\bar{x}_3)$ sets. These calculations will produce the respective lists of "$y_1\, y_2\, y_3\, y_4$" and "y_6."

A very different procedure will be used when the FIND-2 algorithm is calculating the $Y_e(\bar{x}_4)$ set. This is because the hash-search on the "$\bar{x}_4 \cdot a_2$" key will find no corresponding $H_{b_2}(23)$ pointer. The default response of the FIND-2 algorithm will be to inform the user that "$Y_e(\bar{x}_4)$ is an empty set."

Let us conclude this example by assuming that the user asks the FIND-2 algorithm to delete the $\bar{y}_1$ record. In this case, a hash on "$\bar{y}_1 \cdot b_2$" will be used to locate the $H_{b_1}(21)$ pointer. Note that this pointer contains the address of $Q_{e^*}(21)$. The FIND-2 algorithm will therefore make a subroutine call to FIND-1. The latter procedure will be instructed to delete $\bar{y}_1$ from the $Q_{e^*}(21)$ structure. Obviously, all subsequent retrieval operations will reflect this deletion. □

Let us recall the fact that the E-3 expressions were defined in the last chapter as that class of $e(x, y)$ expressions which are built out of several xy-equality predicates concantenated by "AND," "OR" and "NOT" symbols. The FIND-3 algorithm will be designed to produce the $Y_e(x)$ sets for such E-3 expressions in $\Theta(1 + I_e(x); 1)$ runtime. The remainder of this section will be devoted to this algorithm.

The intuitive concept behind the FIND-3 algorithm can be understood if we note that every E-3 predicate can be written as a disjunction of several E-2 predicates similar to equation 3. In that equation, $e(x, y)$ denotes the initial E-3 predicate, and $e_1\, e_2 \cdots e_L$ denote L distinct E-2 predicates:

(3) $$e(x, y) = \{e_1(x,y) \text{ OR } e_2(x,y) \text{ OR } \cdots e_L(x,y)\} \quad .$$

The intuitive idea behind the FIND-3 algorithm is very simple. This procedure will calculate the $Y_e(x)$ sets of E-3 expressions by first having the FIND-2 algorithm process the preceeding e_i predicate and by subsequently taking the union of their $Y_{e_i}(x)$ sets. A more complete description of the FIND-3 algorithm is given below:

<u>Algorithm 5.5.G.</u> Let $e(x, y)$ denote an E-3 expression, and let us assume that equation 3 denotes its decomposition into E-2 parts. The FIND-3 algorithm will be initially activated when the user asks the compile module to construct a database for the processing of predicate of $e(x, y)$. At

that time, this compile module will begin by decomposing $e(x, y)$ into a disjunction of E-2 predicates (similar to the above equation 3) and it will subsequently instruct the FIND-2 compiler to construct each of the L distinct $Q\text{-}2(e_i, R_y)$ data-structures which are associated with the L members of the right-hand side of equation 3. The union of these $Q\text{-}2(e_i, R_y)$ data-structures will constitute the database that the FIND-3 algorithm will use for predicate $e(x, y)$. The data-retrieval and data-modification modules of the FIND-3 algorithm will use this database in the mostly straight-forward manner to perform their respective operations. These two modules are described below:

1) The retrieval module will use a very simple two-step procedure for constructing the user's $Y_e(x)$ sets. The first part of this procedure will consist of subroutine-calls where the FIND-2 algorithm is instructed to search the various $Q\text{-}2(e_i, R_y)$ databases that are associated with equation 3's $e_i(x, y)$ predicates. The purpose of this step will be to produce the $Y_{e_1}(x), Y_{e_2}(x) \cdots Y_{e_L}(x)$ sets. The second step will take the union of these L sets. That union will constitute the user's requested $Y_e(x)$ set.

2) The modification module of the FIND-3 algorithm will be designed to update the various $Q\text{-}2(e_i, R_y)$ structures whenever the user gives an insertion or deletion command. This module will operate in the obvious manner. It will thus make subroutine calls to the FIND-2 algorithm. The latter procedure will be instructed to update each of equation 3's

$Q\text{-}2(e_i, R_y)$ structures.

THEOREM 5.5.H. The FIND-3 algorithm will

I) Correctly construct the $Y_e(x)$ sets of all E-3 expressions.

II) And perform this operation in $\Theta(1 + l_e(x); 1)$ time.

Proof of Proposition I: Note that the procedure in the FIND-3 algorithm was built around the assumption that it is always possible to use an equation similar to 3 for decomposing an E-3 predicate into E-2 predicates. The validity of this assumption can be seen if we write the $e(x, y)$ predicate in the standard disjunction-normalized form. In that case, $e(x, y)$ will appear in precisely the form predicted by equation 3.

The remainder of the proof is trivial. Note that the FIND-3 algorithm utilized equation 3 in the obvious manner. It thus made those subroutine-calls to the FIND-2 algorithm that were suggested by this equation. Also, note that Theorem 5.5.B affirmed the correctness of the FIND-2 algorithm. The correctness of the FIND-3 algorithm thus follows. Q.E.D.

Proof of Proposition II: Once again, our proof will rest on the decomposition suggested by equation 3. Let L denote the number of E-2 terms appearing in that equation. In that case, the retrieval and modification modules of the FIND-3 algorithm will make L subroutine calls to the corresponding modules of the FIND-2 algorithm. The runtime of the FIND-3 algorithm will therefore be approximately L times greater than that of the FIND-2 algorithm.

The remaining proof follows from two simple fact. The first is that

Theorem 5.5.E showed that the FIND-2 algorithm had an $\Theta(1 + I_e(x); 1)$ runtime. The second is that L can be regarded as a coefficient since its value is unrelated to the size of the database. The FIND-3 procedure will therefore also have an $\Theta(1 + I_e(x); 1)$ runtime (where the coefficient is approximately L times larger than that of FIND-2).

Q. E. D.

COROLLARY 5.5.I. It is possible to use the principles in this section to design an $\Theta(1 + I_e(x); 1)$ FIND-3 algorithm whose retrieval module will operate in $2\, I_e(\bar{x})$ units of time (after initial expenditure of $\Theta(1)$ overhead time).

Proof: Note that the FIND-3 algorithm processed E-3 predicates by decomposing them into a disjunction of E-2 predicates. In this proof, it will be once again useful to study that disjunction process. Equation 3 is therefore reproduced below

(3) $$e(x, y) = \{e_1(x, y) \text{ OR } e_2(x, y) \cdots e_L(x, y)\} \quad .$$

There will generally be several different possible series of e_i-predicate that can be substituted into equation 3. Note that the $e_1 e_2 \cdots e_L$ predicates appearing in that equation can be chosen to be fully disjoint. In other words, the predicates can be chosen so that no xy-ordered pair can simultaneously satisfy two or more of these $e_1 e_2 \cdots e_L$ predicates. In that case, the predicates in equation 3 must satisfy equation 4:

(4) $$I_e(x) = I_{e_1}(x) + I_{e_2}(x) + \cdots I_{e_L}(x) \quad .$$

The remainder of the proof is quite simple. Note that Theorem 5.5.E stated that the FIND-2 algorithm would produce its $Y_{e_i}(x)$ sets in $2\, I_{e_i}(x)$ time after overhead operations. This fact together with equation 4 implies that the full $Y_e(x)$ set can be constructed in $2\, I_e(x)$ time (after overhead operations).

Q. E. D.

THEOREM 5.5.J. This chapter's FIND-1, FIND-2, and FIND-3 algorithm will literally possess the best possible runtime magnitudes. Thus, only the coefficient of their $\Theta(1 + I_e(x); 1)$ runtime can be further improved.

Proof: Clearly all algorithms that produce $Y_e(x)$ sets will need a minimum of 1 unit of time for overhead operations, a minimum of $I_e(x)$ units of time to write the $Y_e(x)$ set in the user's work space, and a minimum of one unit of runtime to execute the user's insertion and deletion commands. Thus, all conceivable procedures will require at least $\Theta(1 + I_e(x); 1)$ total time to answer the user's queries. The algorithm in this chapter thus had optimal runtime magnitudes.

Q. E. D.

Remark 5.5.K. Several methods that will improve the runtime coefficient of the FIND-1 algorithm were discussed in Remark 5.4.N of the previous section. All these optimizations obviously apply to the FIND-2 and FIND-3 algorithms (since the latter two algorithms made subroutine-calls to FIND-1). There are also three additional coefficient-optimization techniques

that apply exclusively to the FIND-3 algorithm. These three optimization methods were omitted in the earlier parts of this section because they would have confused the proofs. A brief description of these three coefficient-optimization-strategies is given below:

i) Let $e_1 e_2 \cdots e_K$ denote several distinct E-1 expressions. An E-3 expression will be said to belong to the NEG-1A subclass if it is a predicate of the form:

(5) $$\{[\text{NOT } e_1(x,y)] \text{ AND } [\text{NOT } e_2(x,y)] \text{ AND} \cdots [\text{NOT } e_K(x,y)]\} \quad .$$

Also, let the symbols: $e_A(x,y)$ and $e_B(x,y)$ respectively denote an NEG-1A and E-1 expression. A predicate $e(x,y)$ will be said to be an E-2A expression if it can be written in the form:

(6) $$e(x,y) = \{e_A(x,y) \text{ AND } e_B(x,y)\} \quad .$$

The NEG-1A and E-2A subclasses are important because the respective FIND-1 and FIND-2 algorithms can be expanded so that they will process these two subclasses. Such modifications are important because they will reduce the runtime coefficient of the FIND-3 algorithm. The relevant modifications are completely trivial to implement. Their discussion was omitted in this section only because it would have entailed more complicated

notation.

2) The second coefficient-optimization technique involves the decomposition equation that converts an E-3 predicate into a disjunction of E-2 predicates. There will generally be several equivalent disjunctions of E-2 (or E-2A) expressions that can be used in this step. The runtime coefficient will be improved if the best possible decomposition sequence is picked. The following two rules are useful for choosing an optimized sequence

i) The runtime coefficient of the data-modification module will be improved if there are fewer terms in the disjunction equation and if these terms have a lower complexity.

ii) The runtime coefficient of the data-retrieval-module will be improved if the disjunction equation has fully disjoint predicates.

3) The third coefficient optimization methods involves removing certain redundancies that sometimes occur in FIND-3 database. Obviously, the runtime coefficient will be improved if all such redundancies are removed.

Finally, it should be once again stated that a more detailed discussion of the

above runtime coefficients was omitted from this thesis in the interest of a briefer presentation.

6.1 Chapter Overview

This chapter will constitute the main chapter of this thesis. Most of the claims that were made at the beginning of this thesis will be proven in this chapter. Our discussion will include the SUM algorithm, the COUNT algorithm, the FIND algorithm, the algorithms that evaluate universal and existential quantities, and the algorithms that calculate mean-values, median values, minimum-values and maximum-values. All these algorithms will be proven to manipulate E-7 expressions in those runtimes that were mentioned at the beginning of this thesis.

The discussion in this chapter will be divided into six parts. Section 6.2 will introduce the basic data structures that are employed in this thesis. Section 6.3 will explain how these data structures can be utilized by the "E-4 subclass" of E-7 expression when SUM, FIND and COUNT queries are being performed. Section 6.4 will generalize these three algorithms for the E-5, E-6 and E-7 classes of predicates. Section 6.5 will discuss the further algorithms that evaluate universal quantifiers, existential quantifiers, and mean-values for E-7 expressions. Section 6.6 will discuss the minor generalizations which are needed when processing "K-variabled E-7 expressions." Section 6.7 will discuss the further techniques that are needed to locate the maximum, minimum, median and general i-th smallest element of the subset of y-records that satisfy an E-7 predicate.

The discussion in this section will follow a format which is similar to that which was used in the previous chapters. The main goal will be to produce "dynamic algorithms" that operate in an environment where records are continually being inserted and deleted.

6.2 The Concept of A Data-Image

The main data-structures that will be used in this chapter will be called "data-images". The purpose of this section will be to introduce these data-structures and to explain the notation that will be used to describe them.

Let $e(x, y)$ denote an E-3 expression. The simplest type of data-image will be the $Q^F\text{-}3(e, R_y)$ image. This data-image will consist of the union of those data-structures which are produced by the SUM-3, COUNT-3 and FIND-3 algorithms when they are preparing to calculate the $S_e^F(x)$, $I_e(x)$ and $Y_e(x)$ terms for predicate $e(x, y)$ and relation R_y. Note that the theorems from Chapters 4 and 5 imply that if $e(x, y)$ is an E-3 expression then the $Q^F\text{-}3(e, R_y)$ data-image will support

i) SUM-3 retrieval operations that have an $\Theta(1)$ runtime

ii) COUNT-3 retrieval operations that have an $\Theta(1)$ runtime

iii) FIND-3 retrieval operations that have an $\Theta(1 + I_e(x))$ runtime

iv) insertion and deletion operations that can be performed in $\Theta(1)$ runtime.

The $Q^F\text{-}3(e, R_y)$ data-image will be very centrally related to most of the other data structures that are discussed in this chapter. During that discussion, the symbols of $R_{u(y)}$ and R_{y_v} will be used. The former symbol will denote the subset of the R_y relation that satisfies the $u(y)$ unary condition. The latter symbol will presume that v denotes an interior node of

a tree whose leaves consist of records from R_y. Under these circumstances, R_{y_v} will denote the subset of R_y that is a descendant of v in this tree.

Also, the symbols of $Q^F\text{-}3(e, R_{u(y)})$ and $Q^F\text{-}3(e, R_{y_v})$ will be used in this chapter. These symbols will denote those E-3 data-images that describe the respective $R_{u(y)}$ and R_{y_v} relations relative to the predicate of $e(x, y)$.

The most important symbol in this chapter will be $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$. This symbol will only be used in those applications where $e(x, y)$ denotes an E-3 expression and $b_1 b_2 \cdots b_d$ denotes some attributes of the R_y relation. When the preceeding conditions hold the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-structure will be inductively defined in terms of how many $b_1 b_2 \cdots b_d$ terms it possesses. That inductive definition of $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ will be divided into two parts. The first part (Definition 6.2.A) will define the special data-structure which should be assigned to the $Q^F\text{-}3(e, R_y; b_1)$ data-image. The second part of this definition (which is 6.2.B) will use inductive arguments to define the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-image in terms of the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_{d-1})$ data-image. Throughout this discussion, the $b_1 b_2 \cdots b_d$ terms will be called the "Keys" of $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$. The two part inductive definition of this concept is given below:

Definition 6.2.A. Let $e(x, y)$ denote an E-3 expression and b_1 denote

an attribute of R_y. The $Q^F\text{-}3(e, R_y; b_1)$ data-image will be defined to be that Super-B-Tree which

i) has sorted the elements of R_y by order of increasing $y.b_1$ value;

ii) and satisfies the additional condition that each interior node, v, of this super-B-tree will be assigned an SDS structure that is a $Q^F\text{-}3(e, R_{y_v})$ data-image.

Definition 6.2.B. Let $e(x, y)$ denote an E-3 expression and $b_1 b_2 \cdots b_d$ denote some attributes of R_y. Let us further make the inductive assumption that the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_{d-1})$ data-images are previously defined. Under these circumstances, the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-image will be defined to be that Super-B-Tree which

i) has sorted the elements of R_y by order of increasing $y.b_d$ value

ii) and satisfies the additional condition that the SDS-structure of node v will be a $Q^F\text{-}3(e, R_{y_v}; b_1 b_2 \cdots b_{d-1})$ data-image.

Definition 6.2.C. A $Q^F\text{-}3(e, R_y, b_1 b_2 \cdots b_d)$ data-image will be said to have a degree of d if it contains precisely d distinct $b_1 b_2 \cdots b_d$ Keys.

Remark 6.2.D. The preceeding definition obviously implies that $Q^F\text{-}3(e, R_y)$ is a degree-zero data-image (since it contains no Keys). Sometimes the symbol $Q^F\text{-}3(e, R_y; \phi)$ will be used to denote this data-image. The latter symbol

will appear on those occasions when we wish to emphasize that this data-image has an "empty set" of Keys.

THEOREM 6.2.E. Let $e(x, y)$ denote an E-3 expression and $b_1 b_2 \cdots b_d$ denote d attributes of the R_y relation. Under these circumstances, a y-record can be inserted into or deleted from the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-image in $\Theta(\log^d N)$ time.

The principle of induction will be used to prove the preceeding theorem. The inductive proof will be divided into the standard two parts which examine the cases where $d = 0$ and where d is incrementally increased. These two parts of the proof are given below:

Proof in case where $d = 0$: The proof in this case will largely follow from the theorems that were discussed in Chapters 4 and 5. Those theorems showed that the SUM-3, FIND-3 and COUNT-3 algorithms would support $O(1)$ insertion and deletion operations. Note that the $Q^F\text{-}3(e, R_y; \emptyset)$ data-image is defined to be the union of those structures which these three algorithms produced for the predicate of $e(x, y)$. The previous theorems thus imply that $\Theta(1)$ insertion and deletion operations are possible in the $Q^F\text{-}3(e; R_y; \emptyset)$ data-image. The theorem has thus been verified for the case where $d = 0$.

Q. E. D.

Proof of Inductive Case: The proof in this case follows mostly from the definition of the $Q^F(e, R_y; b_1 b_2 \cdots b_d)$ data-image. Note that this data-image was

defined to be a Super-B-Tree whose SDS substructures were $Q^F\text{-}3(e, R_{y_v}; b_1 b_2 \cdots b_{d-1})$ structures. Also, note that the inductive hypothesis allows us to presume that $\Theta(\log^{d-1} N)$ insertion and deletion operations are possible in these SDS structures. Chapter 3's Super-B-Tree theorem is thus applicable to this data-structure. That theorem implies that the total cost of performing the needed insertion and deletion operations can be computed by multiplying $\log N$ times the amount of time needed to modify the SDS structure. This product will clearly equal $\Theta(\log^d N)$ in the case of $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$. Hence, the theorem has been proven since this data-structure has been shown to possess the required modification-time.

Q. E. D.

COROLLARY 6.2.F. Let $e(x, y)$ once again denote an E-3 expression and $b_1 b_2 \cdots b_d$ once again denote some attributes of the R_y relation. Let us also assume that the R_y relation has N members. Under these circumstances, the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-image

I) Can be constructed in $\Theta(N \log^{D(e)} N)$ time;

II) And will occupy no more than $\Theta(N \log^{D(e)} N)$ space in computer memory.

Proof of Proposition I. Note that Theorem 6.2.E indicated that any given y-record can be inserted into the $Q^F\text{-}3(e, R_y; b_1 b_2 \cdots b_d)$ data-image in $\Theta(\log^{D(e)} N)$ time. It consequently follows that all N y-records can be inserted into this data-structure in $\Theta(N \log^{D(e)} N)$ time. The latter quantity

represents the amount of time needed to construct the complete $Q^F\text{-}3(e, R_y; b_1\, b_2 \cdots b_d)$ data-image. Q. E. D.

Proof of Proposition II. Note that a construction program which operates in $\mathcal{O}(N \log^{D(e)} N)$ time cannot possibly build a data-structure that occupies more than $\mathcal{O}(N \log^{D(e)} N)$ space. This reasoning combined with Proposition I implies that the $Q^F\text{-}3(e, R_y; b_1\, b_2 \cdots b_d)$ data-image can be stored in $\mathcal{O}(N \log^{D(e)} N)$ space. Q. E. D.

Comment: Let $u_i(y)$ denote some unary predicate, $R_{u_i(y)}$ denote the subset of R_y that satisfies this predicate, and $Q^F\text{-}3(e, R_{u_i(y)}; b_1\, b_2 \cdots b_d)$ denote the corresponding data-image which describes the $R_{u_i(y)}$ relation. In the later parts of this chapter, it will be proven that all E-7 predicates can be efficiently processed by using data-structures that consist of unions of several such $Q^F\text{-}3(e, R_{u_i(y)}; b_1\, b_2 \cdots b_d)$ data-images. This fact is important because it will imply that the previous two theorems can also describe the amount of data-modification time and storage space that is consumed by the more sophisticated data-structures of this chapter.

Remark 6.2.G. In most applications, the FIND-3, COUNT-3 and SUM-3 algorithms will produce databases for a given E-3 predicate that contain common data-substructures. Note that the $Q^F\text{-}3(e, R_y)$ data-image was

defined to be the union of the three databases which these algorithms produce for the $e(x, y)$ predicate. It will always be assumed in this chapter that this union-taking operation is sufficiently sophisticated to adjust the $Q^F\text{-}3(e, R_y)$ data-image so that it does not contain two or three copies of the same data-substructure. Such a more compact representation of the $Q^F\text{-}3(e, R_y)$ data-image is desirable because it will reduce the runtime coefficient by eliminating unnecessary repetions. This type of representation for the $Q^F\text{-}3(e, R_y)$ data-image is important because Definitions 6.2.A and 6.2.B indicated that the higher-level $Q^F\text{-}3(e, R_y; b_1\, b_2 \cdots b_d)$ data-image will contain many such "degree zero" $Q^F\text{-}3(e, R_y)$ data-substructures. The runtime coefficients for the higher-level $Q^F\text{-}3(e, R_y; b_1\, b_2 \cdots b_d)$ algorithms will thus be improved if the degree-zero data-images are given this more compact representation.

During this chapter, the symbols of $Q^F\text{-}4(e, R_y)$, $Q^F\text{-}5(e, R_y)$, $Q^F\text{-}6(e, R_y)$ and $Q^F\text{-}7(e, R_y)$ will be frequently used. These symbols will denote the more sophisticated data-images which are employed by the E-4 through E-7 expressions. The definition of these four types of data-images will be given in the next two sections of this chapter. A more intuitive explanation of the meanings of these terms will be given in the next several paragraphs.

Throughout this chapter, the symbol of "$Q^F\text{-}K(e, R_y)$" will be defined only when $e(x, y)$ belongs to the E-K class of expressions (for

$K = 3, 4, 5, 6$ or 7). When defined, $Q^F\text{-}K(e, R_y)$ will represent that data-structure which enables the SUM, COUNT and FIND algorithms to query predicate e with the runtimes that were mentioned in Chapter 1.

One of the themes in this chapter will be that each class of E-K predicates and $Q^F\text{-}K(e, R_y)$ data-images will be a generalization of their E-(K-1) and $Q^F\text{-}(K-1)\text{-}(e, R_y)$ predecessors. Consequently, the definition of the final $Q^F\text{-}7(e, R_y)$ data-image structure will possess all the characteristics of the earlier data-images as well as some additional features.

In some parts of this chapter, it will be convenient to omit the hyphenated number symbol (as in "$Q^F(e, R_y)$"). Such hyphenated numbers will be omitted in those contexts where no distinction is being made between the E-1, E-4, E-5, E-6 and E-7 expressions. In such cases, the relevant theorems will hold for all classes of data-images.

6.3 The Basic E-4 Algorithms

This section will explain how the data-images from the previous sections can enable the SUM-4, FIND-4 and COUNT-4 algorithms to efficiently process "E-4" expressions. The ideas in this section are important because they will be later used by the E-5, E-6, and E-7 algorithms. The discussion will now begin with an important definition.

Definition 6.3.A. A predicate, $r(x, y, b)$ will be said to be an xy-range-term with an attribute of b if all of its atomic predicates are order comparisons between $y.b$ and some attribute of x.

Example 6.3.B. Equations 1 through 4 are some examples of range-terms with an attribute of b_1:

1) $\{x.a_1 \leq y.b_1\}$

2) $\{x.a_1 < y.b_1 \text{ AND } x.a_2 > y.b_1 \text{ AND } x.a_3 > y.b_1\}$

3) $\{x.a_1 \leq y.b \text{ OR } x.a_2 \geq y.b_1 \text{ OR } x.a_3 > y.b_1\}$

4) $\{\text{NOT } x.a_1 \leq y.b_1\}$

5) $\{x.a_1 \cdot y.b_1 \text{ OR } [x.a_2 > y.b_1 \text{ AND NOT } x.a_3 < y.b_1]\}$.

Definition 6.3.C. A predicate will be said to be an E-4 expression if it is either an E-3 expression or a conjunction of several range-terms with an E-3 expression.

Definition 6.3.D. Let $e^*(x, y)$ denote an E-3 expression and let

$r_1(x, y.b_1)$, $r_2(x, y.b_2) \cdots r_d(x, y.b_d)$ denote a collection of d range-terms. Let us examine the prototype E-4 expression given in equation 6:

6) $e(x,y) = \{e^*(x,y) \text{ AND } r_1(x,y.b_1) \text{ AND } r_2(x,y.b_2) \text{ AND} \cdots r_d(x,y.b_d)\}$

$e^*(x, y)$ will be henceforth called the "E-3 component" of this expression. Also, $b_1 b_2 \cdots b_d$ will be called the "Keys" of this expression. Note that equation 6 possessed d Keys. This equation will thus be said to have a "degree equal to d." The symbol of "$D(e)$" will henceforth denote the degree of predicate e.

Example 6.3.E. One example of an E-4 expression is given in equation 7:

7) $\{x.a_1 = y.b_1 \text{ AND } (x.a_1 < y.b_2 \text{ OR } x.a_2 < y.b_2) \text{ AND } x.a_3 < y.b_3\}$.

Note that the E-3 component of that expression equaled "$x.a_1 = y.b_1$" and that its two Keys were b_2 and b_3. This expression thus had a degree equal to 2.

A second example of an E-4 expression is given in equation 8. Note that this equation differs from 7 insofar as its $y.b_3$ attribute has been changed to a third $y.b_2$ attribute. This equation thus has a degree of one because the entire bracketed expression can be regarded as a single range-term:

8) $\{x.a_1 = y.b_1 \text{ AND } [(x.a_1 < y.b_2 \text{ OR } x.a_2 < y.b_2) \text{ AND } x.a_3 < y.b_2]\}$.

Two rather unusal examples of E-4 expressions are shown in equations 9 and 10. Note that equation 9 is an E-4 expression because "TRUE" is an E-3 expression. Equation 10 can be regarded to be an E-4 expression because it is obviously equivalent to 9

9) $\{\text{TRUE AND } x.a < y.b\}$

10) $\{x.a < y.b\}$.

Data-Image 6.3.F. Let $e(x, y)$ denote an E-4 expression whose E-3 component is $e^*(x, y)$ and whose Keys are $b_1 b_2 \cdots b_d$. Under these circumstances, the symbol of $Q^F\text{-}4(e, R_y)$ will signify that data-image which previously has been denoted as $Q^F(e^*, R_y; b_1 b_2 \cdots b_d)$.

Comment: The rest of this section will show that the $Q^F\text{-}4(e, R_y)$ data-image will enable the SUM, FIND, COUNT, insertion and deletion algorithms to operate in $\Theta(\log^{D(e)} N)$ time.

LEMMA 6.3.G. A y-record may be inserted into or deleted from the $Q^F\text{-}4(e, R_y)$ structure in $\Theta(\log^{D(e)} N)$ worst-case hashtime. Also, this data-image will occupy no more than $\Theta(N \log^{D(e)} N)$ storage space.

Proof: Lemma 6.3.G can be regarded as a restatement of the results which were proven earlier in Theorem 6.2.E and Corollary 6.2.F. The only difference between these propositions is the notation which they use. Lemma 6.3.G is thus a consequence of the earlier propositions and of the

definitions of $D(e)$ and of $Q^F\text{-}4(e, R_y)$. Q. E. D.

Definition 6.3.H. Let $r(x, y, b)$ denote a range term, T denote a tree whose y-leaves are arranged in increasing y, b order, and v denote an interior node of T. v will be said to be a critical node with respect to r, T, and $\bar{x}$ if the following two conditions hold:

i) All descendant y-leaves of v must satisfy the condition of $r(\bar{x}, y, b)$.

ii) Also, the same must not be true for the father of v. In other words, some descendant leaf of that father must fail to satisfy $r(\bar{x}, y, b)$.

The concept of a critical node will play a major role throughout this section. It will be used in the description of the SUM-4, FIND-4 and COUNT-4 algorithms. This discussion will now begin with SUM-4:

Algorithm 6.3.H. Let $e(x, y)$ denote an E-4 expression whose E-3 component is $e^*(x, y)$, whose Keys are $b_1 b_2 \cdots b_d$, and whose range-terms are $r_1(x, y, b_1), r_2(x, y, b_2) \cdots r_d(x, y, b_d)$. The SUM-4 algorithm will be designed to scan the $Q^F\text{-}4(e, R_y)$ data-image for the purposes of calculating an $S_e^F(\bar{x})$ quantity. The procedure used by the SUM-4 algorithm will depend on the degree of the E-4 expression which is being processed. If $D(e) = 0$ then a trivial subroutine call to the SUM-3 algorithm will calculate $S_e^F(\bar{x})$ (since $e(x, y)$ is an E-3 expression when

such a degenerate degree prevails). Otherwise, the required calculation will be performed by the following 3-step procedure:

1) The first step will take advantage of the fact that the $Q^F\text{-}4(e, R_y)$ data-image is a Super-B-Tree whose y-leaves are arranged in order by increasing y, b_d value. This step will locate those interior nodes of this Super-B-Tree which are critical relative to the $r_d(\bar{x}, y, b_d)$ condition. (Critical nodes were defined in the previous paragraph.)

2) Let v denote a critical node located by the previous step. Let $S_e^F(\bar{x}, v)$ denote the sum of those y-records which satisfy $e(\bar{x}, y)$ and are simultaneously a descendant of node v. This step will calculate the $S_e^F(\bar{x}, v)$ quantities for every one of $r_d(\bar{x}, y, b_d)$'s critical nodes. A recursive call to the SUM-4 algorithm will be used to perform these calculations. The details of these recursive calls can be best explained if more notation is introduced. Let R_{y_v} denote the subset of y-records that are a descendant of node v, and $e^\#(x, y)$ denote the predicate expression defined below

ii) $e^\#(x,y) = \{e^*(x,y) \text{ AND } r_1(x,y,b_1) \text{ AND } r_2(x,y,b_2) \text{ AND} \cdots r_{d-1}(x,y,b_{d-1})\}$.

This step will calculate the $S_e^F(\bar{x}, v)$ values by having the SUM-4 algorithm recursively call itself and by instructing the invoked

procedure to search v's SDS structure with the $e^{\phi}(\bar{x}, y)$ predicate. (The reason such a procedure operates correctly is that the definition of $Q^F\text{-}4(e, R_y)$ data-images implies that v's SDS structure is a $Q^F\text{-}4(e^{\phi}, R_{y_v})$ data-image.)

3) The third step will calculate the value of $S_e^F(\bar{x})$ by taking the sum of all the $S_e^F(\bar{x}, v)$ quantities which are associated with $r_d(\bar{x}, y.b_d)$'s critical nodes.

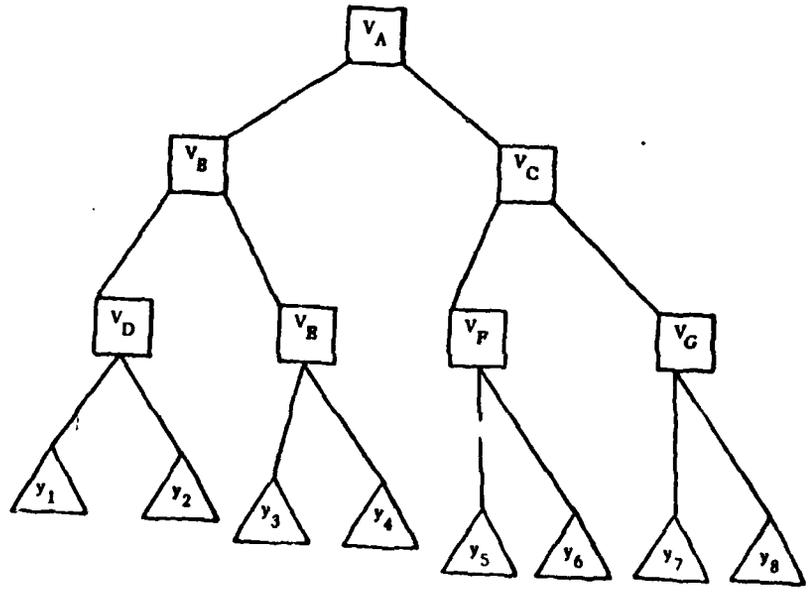
Example 6.3.1. Let us watch the SUM-4 algorithm calculate $S_e^F(x)$ values for the following E-4 predicate:

(2) $$e(x, y) = \{x.a_1 = y.b_1 \text{ AND } x.a_2 > y.b_2\} \quad .$$

In this example, it will be assumed that R_y is an eight-element relation whose members are described by the following table:

element-name	$y.b_1$ value	$y.b_2$ value	$F(y)$ value
$\bar{y}_1$	10	21	1.1
$\bar{y}_2$	12	22	1.2
$\bar{y}_3$	10	23	1.3
$\bar{y}_4$	14	24	1.4
$\bar{y}_5$	10	25	1.5
$\bar{y}_6$	16	26	1.6
$\bar{y}_7$	10	27	1.7
$\bar{y}_8$	18	28	1.8

Note that the $Q^F\text{-}4(e, R_y)$ data-image in our example is defined to be a Super-B-Tree whose y-leaves are arranged by order of increasing $y.b_2$ value. One possible pictorial representation of that tree is given below. The leaves and interior-nodes in that diagram are respectively represented with triangles and squares.



The structure of the interior nodes of the preceeding Super-B-Tree can be understood if equation 12 is examined in further detail. Note that the E-3 component of that equation was "$x.a_1 = y.b_1$." It thus follows that the SDS of each node, v, of our Super-B-Tree will be a $Q^F(\text{"}x.a_1 = y.b_1\text{"}; R_{y_v})$ data-image.

Let us watch the SUM-4 algorithm use the preceeding data-structure to process an x-element where $\bar{x}.a_1 = 10$ and $\bar{x}.a_2 = 26.5$. The SUM-4 algorithm will use its normal 3-step procedure for calculating $S_e^F(\bar{x})$. These three steps are described below:

1) The first step will find all the nodes which are "critical" with respect to the element of $\bar{x}$ and the range-term of "$x.a_2 > y.b_2$." Note that v_B and v_F are the critical nodes in this example.

2) The second step of the SUM-4 algorithm will be designed to calculate the $S_e^F(\bar{x}, v)$ subtotals for these two critical nodes. *The details of this procedure can be best explained if it is noted* that our data-structure will cause node v to have a SDS structure of $Q^F\text{-}3(\text{"}x.a_1 = y.b_1\text{"}; R_{y_v})$. Our algorithm will thus make two subroutine-calls to the SUM-3 procedure, and it will instruct the invoked program to search the SDS structures of v_B and v_F with the argument of $\bar{x}$ and the predicate of

"$x.a_1 = y.b_1$." The quantities produced by these searches will be the two required $S_e^F(\bar{x}, v)$ values. The specific value of these two quantities is shown below:

13) $S_e^F(\bar{x}, v_B) = F(\bar{y}_1) + F(\bar{y}_3) = 2.4$

14) $S_e^F(\bar{x}, v_F) = F(y_S) = 1.5$.

3) *The last step of the SUM-4 algorithm will add together the* preceeding two quantities to derive the value of $S_e^F(\bar{x})$.

<u>THEOREM 6.3.K.</u> Let $e(x, y)$ denote an E-4 expression, and let us assume that the $Q^F\text{-}4(e, R_y)$ data-image is physically constructed. Under these circumstances, Algorithm 6.3.1 will

I) Correctly calculate the $S_e^F(x)$ sums for E-4 expressions;

II) And perform this task in $\mathcal{O}(\log^{D(e)} N)$ time.

<u>Proof of Proposition I.</u> The correctness of the SUM-4 algorithm will be *proven by induction on the degree of the relevant* E-4 *expression. Note that* the theorem is trivial to verify for the case of degree-zero E-4 expressions (since Chapter 4's SUM-3 Theorem can be applied to those degenerate expressions). Our attention will therefore focus on the more difficult inductive case.

The SUM-4 algorithm will execute Steps 1 through 3 when it is inductively processing an $e(x, y)$ predicate that satisfies $D(e) \geq 1$. An

examination of this 3-step procedure reveals that it was based on the two assumptions that

a) All y-records satisfying $e(\bar{x}, y)$ are a descendant of precisely one critical node;

b) And Step 2 of the SUM-4 algorithm will correctly calculate the $S_e^F(\bar{x}, v)$ values of these critical nodes.

The first assumption can be easily verified by the definition of critical nodes. The verification of the second assumption will rest on an examination of the second step of Algorithm 6.3.1. That step calculated the value of $S_e^F(\bar{x}, v)$ through a recursive call where v's SDS structure was searched with the "$e^{\phi}(x, y)$" predicate. Note that $e^{\phi}(x, y)$ has a degree equal to $D(e) - 1$. The inductive hypothesis is thus applicable. It together with the definition of $Q^F\text{-}4(e, R_y)$ confirm that $S_e^F(\bar{x}, v)$ is correctly calculated by Step 2 of this algorithm.

The remaining proof of Proposition I is trivial. Note that statements a and b imply that the sum of the critical nodes $S_e^F(\bar{x}, v)$ values equal $S_e^F(\bar{x})$. This sum is calculated in Step 3 of Algorithm 5.4.1. Our proof has thus verified that the SUM-4 algorithm will correctly calculate this quantity.

Q. E. D.

<u>Proof of Proposition II.</u> It is easiest to verify the second proposition if an inductive mode of proof is used. Note that Proposition II is trivial to prove

for the case of degree-zero E-4 expression. This is because the SUM-3 Theorem (which was Theorem 4.4.1.) implies that such degenerate E-4 expressions can be processed in $\Theta(1)$ retrieval time. The remaining proof will therefore focus on the more complicated inductive case.

The analysis of this case will center around the fact that each $Q^F\text{-}4(e, R_y)$ data-image is defined to be a Super-B-Tree. Let h denote the height of this Super-B-Tree and L denote the number of atomic order-predicates appearing in Algorithm 6.3.1's $r_d(x, y, b_d)$ range-term. Under these assumptions, it is easy to verify that the $Q^F\text{-}4(e, R_y)$ Super-B-Tree will never possess more than Lh critical nodes. The preceeding implies that:

a) Steps 1 and 3 of Algorithm 6.3.1 will never take more than $\Theta(Lh)$ runtime.

b) There will never be more than Lh recursive invocations of the SUM-4 algorithm made by Step 2.

The cost of all of Step 2's recursive invocations can be calculated with the help of the inductive hypothesis. Note that each recursive invocation of the SUM-4 algorithm consisted of a search with the "$e^{\phi}(x, y)$ predicate." Also, note this predicate had a degree equal to $D(e) - 1$. The inductive hypothesis thus allows us to presume that each recursive invocation will consume $\Theta(\log^{D(e)-1} N)$ time. All of Step 2's invocations will thus consume never more than $\Theta(Lh \cdot \log^{D(e)-1} N)$ time.

The remainder of the proof will follow from Chapter 3's Super-B-Tree Theorem. That theorem stated that all Super-B-Trees have $\Theta(\log N)$ heights. This value of h clearly implies that the total time consumed by Steps 1 through 3 of the SUM-4 algorithm has an "$L \cdot \log^{D(e)} N$" magnitude. Also, note that L was defined to be equal to the number of atomic predicates in the r_d range-term. L is thus a runtime coefficient (since its value is unrelated to the size of the database). The proof is thus completed since the SUM-4 algorithm has been shown to have an $\Theta(\log^{D(e)} N)$ runtime.

Q. E. D.

COROLLARY 6.3.L. Let us use the terms of "COUNT-4" and "FIND-4" to refer to those procedures which are identical to SUM-4 except that the former two procedures will make subroutine-calls to the COUNT-3 and FIND-3 algorithms in those places where SUM-4 called SUM-3. Let us also assume that $e(x, y)$ is an E-4 expression and that the user has previously constructed the corresponding $Q^F\text{-}4(e, R_y)$ data-image. Under these circumstances, it will follow that:

I) The COUNT-4 retrieval algorithm can calculate any $I_e(\bar{x})$ quantity in $\Theta(\log^{D(e)} N)$ time;

II) The FIND-4 retrieval algorithm can construct any $Y_e(\bar{x})$ set in $\Theta(I_e(\bar{x}) + \log^{D(e)} N)$ time (where the $I_e(x)$ term is qualified by a runtime coefficient of 2).

Proof: Note that the COUNT-4 and FIND-4 algorithms used procedures that were virtually identical to SUM-4. It thus follows that the two claims that were made in this corollary can be verified with reasoning similar to that which was used in the proof of Theorem 6.3.K.

Q. E. D.

THEOREM 6.3.M. Let $e(x, y)$ denote an E-4 expression whose data-image is $Q^F\text{-}4(e, R_y)$. The preceeding theorems have shown that this data-structure will occupy no more than $\Theta(N \log^{D(e)} N)$ storage space and that it will support:

I) A SUM-4 algorithm which calculates $S_e^F(x)$ in $\Theta(\log^{D(e)} N)$ time;

II) A COUNT-4 algorithm which calculates $I_e(x)$ values in $\Theta(\log^{D(e)} N)$ time;

III) A FIND-4 algorithm which calculates $Y_e(x)$ sets in $\Theta(I_e(x) + \log^{D(e)} N)$ time (with $I_e(x)$ qualified by a runtime coefficient of 2).

IV) Insertion and deletion operations which modify the $Q^F\text{-}4(e, R_y)$ data-image in $\Theta(\log^{D(e)} N)$ time.

The preceeding theorem is of course a summary of the results which were proven in Propositions 6.3.G, 6.3.K, and 6.3.L. This section's SUM-4, COUNT-4 and FIND-4 algorithm are important because they will perform most of the work for the more advanced E-5, E-6 and E-7 expressions.

Remark 6.3.N. The preceeding discussion of E-4 expressions was deliberately kept as brief as possible. There are several optimizations which can be added to the E-4 algorithms for the purposes of increasing their efficiency. The most important of these optimizations would require us to slightly change Definition 6.2.A. That part of this chapter gave the definition of the "degree-one" data-images such as the $Q^F(e, R_y; b_1)$ prototype. This definition indicated that these data-images were Super-B-Trees where every vertice, v, of these trees was assigned an SDS structure that consisted of a $Q^F\text{-}3(e, R_{y_v})$ data-image. The improvements that are possible can be understood if it is noted that the definition of $Q^F\text{-}3(e, R_{y_v})$ implies that this data-structure will contain a hashlist that describes the R_{y_v} subrelation. This hashlist was used by the FIND-3 algorithm during its retrieval operations. It is possible to revise the FIND-3 algorithm so that it does not need to employ this hashlist. This is because the FIND-3 algorithm can locate the elements of R_{y_v} by using an alternate search procedure that seeks to find the y-leaves which are a descendant of node v. If such an alternate procedure is used then it would be possible to remove the R_{y_v} hashlist from v's SDS structure without any resulting harm being done to the retrieval time of our basic search algorithms. Such changes would enable the $Q^F\text{-}3(e, R_y; b_1)$ data-image to attain a more compact form that saves memory space and data-modification time. These changes are important because the higher-level $Q^F\text{-}4(e, R_y)$ and $Q^F\text{-}3(e, R_y; b_1\, b_2 \cdots b_d)$ data-images are defined to

contain such "degree-one" data-images. The modified definition of the degree-one structures would thus produce savings in memory space and data-modification time for all the data-images in this chapter. It must be emphasized that the proposed changes would improve utilization of memory space and data-modification time without impairing retrieval time. Formal theorems and proofs about this topic were omitted from this thesis so that the subject matter could be given a briefer presentation.

Remark 6.3.O. The preceeding optimization is especially important to apply when the user is processing an E-4 predicate whose E-3 component equals "TRUE." In that case, this optimization will store the $Q^F\text{-}4(e, R_y)$ data-image in $\Theta(N \log^{D(e)-1} N)$ space. This amount of storage space is often significantly cheaper than the $\Theta(N \log^{D(e)} N)$ space which an unoptimized algorithm would use.

6.4 The Basic E-5, E-6 and E-7 Algorithms

This section will discuss the versions of the SUM, FIND, and COUNT algorithms that process E-5, E-6 and E-7 expressions. The theorems in this section are important because they will show that the SUM, FIND and COUNT algorithms can operate with the efficiencies that were originally claimed in Chapter 1.

The discussion in this section will focus mainly on the unary predicates. The definition of unary predicates was given in the introduction of this thesis. That section indicated that an unary predicate was defined to be any atomic predicate whose truth-value depends on only one variable.

There will be three kinds of unary predicates discussed in this section. These are the single-attribute, double-attribute, and list-oriented unary predicates. Their definitions are given below:

Definition 6.4.A. A single attribute unary predicate will be defined to be an (equality or order) comparison between an attribute of a variable and a constant. Two examples of such one-attribute predicate are "$x.a = 10$" and "$x.a > 10$." A double-attribute unary predicate will be defined to be an (order or equality) comparison between two attributes of a single variable. Two examples of this type of predicate are "$x.a_1 = x.a_2$" and "$x.a_1 \leq x.a_2$." A list-oriented unary predicate will be defined to be any predicate where the set of elements satisfying this predicate is defined by some user-supplied list. An

example of such a list-oriented expression is that predicate generated by the list of employees working in a given department.

The preceeding three classes of unary predicates will be extremely important because many users will find them convenient to employ. The following three definitions will play a major role in our discussion:

Definition 6.4.B. An unary predicate will be said to be "x-based" if it uses the variable of x. Similarly, an unary predicate will be said to be "y-based" if it uses the variable of y.

Definition 6.4.C. An (usually nonatomic) predicate expression will be said to be an "x-based unary term" if all its atomic predicates are x-based unary predicates. Similarly, a predicate expression will be a "y-based unary term" if all its atomic predicates are y-based unary predicates.

Example 6.4.D. Equations 1 and 2 are examples of x-based unary terms. Equations 3 and 4 are examples of y-based unary terms:

1) $\{x.a_1 = 10 \text{ OR } x.a_2 > 20\}$

2) $\{x.a_1 > x.a_2 \text{ AND } x.a_3 = 10\}$

3) $\{\text{NOT } y.b_1 = 10\}$

4) $\{y.b_1 > 10 \text{ AND } (y.b_2 > 20 \text{ OR NOT } y.b_3 = y.b_4\}$.

The first topic of this section will be the versions of the COUNT, SUM

and FIND algorithms that process E-5 expressions. The definition of E-5 expressions is given below:

Definition 6.4.E. A predicate will be said to be an E-5 expression if it is either an E-4 expression or a conjunction between an E-4 expression and an x-based unary term.

Definition 6.4.F. Let $e^*(x, y)$ denote an E-4 expression, $u(x)$ denote an y-based unary-conjunction term, and $e(x, y)$ denote the prototype E-5 expression given below:

5) $e(x, y) = \{u(x) \text{ AND } e^*(x, y)\}$

$u(x)$ will be henceforth called e's unary component, and $e^*(x, y)$ will be called its E-4 component.

Data-Image 6.4.G. Let $e(x, y)$ denote an E-5 expression whose E-4 component is $e^*(x, y)$. The symbol "Q^F-5(e, R_y)" will denote e's data-image. The definition of this data-image is quite simple. Its data-structure will simply be defined to be identical to that of Q^F-4(e^*, R_y).

Most of the theorems in this section will show that our various data-images support $\Theta(\log^{D(e)} N)$ search operation. In that discussion, "D(e)" will denote the degree of expression e. The general definition of D(e) was given in Chapter 1. That section indicated that D(e) was defined to be equal to the number of different y.b attributes appearing in e's order

predicates.

Let $e(x, y)$ denote an E-5 expression whose E-4 component is $e^*(x, y)$ and whose unary component is $u(x)$. The next several paragraphs will explain how the SUM-5, COUNT-5, FIND-5 and data-modification algorithms can process such an E-5 predicate in $\log^{D(e)} N$ time.

All four of our E-5 algorithms are straightforward generalizations of their E-4 counterparts. Let us begin by examining the SUM-5 retrieval procedure. This algorithm will use a simple two-step program for calculating $S_e^F(\bar{x})$ values. The first part of this program will check to see whether $\bar{x}$ satisfies e's unary component of $u(\bar{x})$. If $\bar{x}$ fails to satisfy $u(\bar{x})$ then $S_e^F(\bar{x})$ will be automatically set equal to zero (because no y-records can satisfy $e(\bar{x}, y)$ in this case). If $\bar{x}$ does satisfy $u(\bar{x})$ then the SUM-5 algorithm will calculate the value of $S_e^F(\bar{x})$ by making a subroutine-call to SUM-4. This subroutine-call will instruct the latter procedure to calculate the value of $S_e^F(\bar{x})$ by searching the Q^F-5(e, R_y) data-image with the $e^*(x, y)$ predicate.

THEOREM 6.4.H. The SUM-5 algorithm (outlined in the previous paragraph) will

I) Correctly calculate the $S_e^F(\bar{x})$ values of E-5 expressions;

II) And perform this operation in $\Theta(\log^{D(e)} N)$ runtime.

Proof of Proposition I. Let $e^*(x, y)$ denote the E-4 component of $e(x, y)$ and $u(x)$ denote its unary compoent. An examination of the SUM-5 algorithm indicates that this algorithm was built around the three assumptions that

A) The value of $S_e^F(\bar{x})$ will equal zero when $\bar{x}$ fails to satisfy $u(\bar{x})$.

B) The value of $S_e^F(\bar{x})$ will equal $S_{e^*}^F(\bar{x})$ when $\bar{x}$ satisfies $u(\bar{x})$.

C) And the assumption that SUM-4 algorithm will correctly calculate the value of $S_{e^*}^F(\bar{x})$ when it is scanning the $Q^F\text{-}5(e, R_y)$ data-image.

All three of these assumptions are easy to verify. The validity of the first two assumptions follows from the definitions of E-5 expressions. The correctness of the third assumption follows from the definition of $Q^F\text{-}5(e, R_y)$ data-images, the definition of E-5 expressions, and the SUM-4 Theorem (which was Proposition 6.3.K).

The preceeding three assumptions are important because they collectively imply the correctness of the SUM-5 algorithm. The validity of the SUM-5 algorithm has thus been proven (since we have verified assumptions A through C).

Q. E. D.

Proof of Proposition II. The runtime of the SUM-5 algorithm is easiest to measure if we separately examine the two steps of this procedure. The first step of this procedure will check to see whether $\bar{x}$ satisfies the $u(x)$ unary component. Clearly, this determination can be made in $\Theta(1)$ time. The second step of the SUM-5 algorithm will consist of a subroutine-call to the SUM-4 algorithm. Note that Theorem 6.3.K indicated that this invocation of the SUM-4 algorithm will consume $\Theta(\log^{D(e)} N)$ runtime. The total runtime consumed by these two steps will thus have an $\Theta(\log^{D(e)} N)$ worst-case magnitude. The SUM-5 algorithm has thus been shown to operate in the required runtime.

Q. E. D.

Example 6.4.I. Let $e(x, y)$ denote the following E-5 expression

6) $e(x, y) = \{x.a_1 = 10 \text{ AND } x.a_2 > y.b_2\}$.

Let $u(x)$ and $e^*(x, y)$ denote the respective unary and E-4 components of equation 6. Note that in this example, $u(x)$ equals "$x.a_1 = 10$" and $e^*(x, y)$ equals "$x.a_2 > y.b_2$."

The SUM-5 retrieval algorithm will use the preceeding components in the obvious manner when it is calculating $S_e^F(x)$ values. Consider the cases of two elements, $\bar{x}_1$ and $\bar{x}_2$, which satisfy the equalities of $\bar{x}_1.a_1 = 5$ and $\bar{x}_2.a_1 = 10$. Under such circumstances, SUM-5 will set $S_e^F(\bar{x}_1)$ equal to zero (because $\bar{x}_1$ fails to satisfy $U(x)$), and it will set $S_e^F(\bar{x}_2)$ equal to $S_{e^*}^F(\bar{x}_2)$ (because $\bar{x}_2$ does satisfy this unary predicate).

COROLLARY 6.4.J. Let us use the terms COUNT-5 and FIND-5 to refer to those procedures which are identical to SUM-5 except that the former two procedures will make subroutine-calls to COUNT-4 and FIND-4 in those places where SUM-5 called SUM-4. Let us also assume that $e(x, y)$ is an E-5 expression and that the user has previously constructed the corresponding Q^F-4(e, R_y) data-image. Under these circumstances, it will follow that:

I) The COUNT-5 algorithm can calculate any $I_e(\bar{x})$ quantity in $\Theta(\log^{D(e)} N)$ time;

II) The FIND-5 algorithm can construct any $Y_e(\bar{x})$ set in $\Theta(I_e(x) + \log^{D(e)} N)$ time (where the $I_e(x)$ term is qualified by a runtime coefficient of 2).

Proof: Clearly, the COUNT-5 and FIND-5 algorithms are sufficiently similar to SUM-5 for the proof of Theorem 6.4.H to be also applicable to these two algorithms. Hence, *our previous reasoning can be used to confirm* the two claims that were made in this corollary.

Q. E. D.

THEOREM 6.4.K. A y-record can always be inserted into or deleted from the Q^F-5(e, R_y) data-image in $\Theta(\log^{D(e)} N)$ runtime.

Proof: Note that Q^F-5(e, R_y) was defined to have a data-structure that was identical to that of Q^F-4(e^*, R_y). It thus follows that the same insertion and deletion algorithms apply to both the E-4 and E-5 data-images. Hence,

Lemma 6.3.G may be used to verify that the E-5 insertion and deletion operations will take $\Theta(\log^{D(e)} N)$ runtime.

Q. E. D.

Comment: Obviously, the collective implication of Theorems 6.4.H, 6.4.J, and 6.4.K is that the Q^F-5(e, R_y) data-image will support $\Theta(\log^{D(e)} N)$ insertion, deletion, SUM-5, COUNT-5 and FIND-5 operations. Thus, this data-image is a very useful structure. We will now develop the generalizations of these propositions for E-6 and E-7 expressions.

Definition 6.4.L. Let $e^*(x, y)$ denote an E-5 expression, and $u(y)$ denote a y-based unary term. A predicate, $e(x, y)$, will be said to be an E-6 expression if it is either an E-5 expression or if it is a conjunction of the form: "$u(y)$ AND $e^*(x, y)$." In the latter case, $u(y)$ and $e^*(x, y)$ will be called the respective y-unary and E-5 components of $e(x, y)$.

Data-Image 6.4.M. Let $e(x, y)$ denote an E-6 expression whose respective y-unary and E-5 components are $u(y)$ and $e^*(x, y)$. Also let $R_{u(y)}$ denote the subset of R_y which satisfies the $u(y)$ condition. The symbol, "Q^F-6(e, R_y)," will henceforth denote the E-6 data-image of $e(x, y)$. The data-structure of this image will be defined to be identical to that of Q^F-5$(e^*, R_{u(y)})$. In other words, the Q^F-6(e, R_y) data-image will be defined to have the same data-structure as that data-image which is produced when the $R_{u(y)}$ *relation is spanned by the* $e^*(x, y)$ *predicate.*

All of the E-6 retrieval algorithms will be trivial extensions of the E-5 procedures. For example, the SUM-6 algorithm will use an extremely simple one-step procedure to calculate $S_e^F(\bar{x})$ values. The single step of this procedure will consist of a subroutine-call to the SUM-5 algorithm. That subroutine-call will instruct SUM-5 to calculate the value of $S_e^U(\bar{x})$ by searching the $Q^F\text{-}6(e, R_y)$ data-image with the $e^*(\bar{x}, y)$ predicate. The FIND-6 and COUNT-6 algorithms will use similar subroutine-calls to the FIND-5 and COUNT-5 algorithms to perform their calculations. The correctness of all three E-6 retrieval algorithms is a trivial consequence of our E-5 theorems and of Definitions 6.4.L and 6.4.M.

A slightly more complicated E-6 algorithm will be used for performing insertion and deletion operations. The E-6 data-modification procedure will have two steps. The first step will check to see whether the specified y-record satisfies u(y). The second step will only be invoked when the preceeding condition is satisfied. In that case, a subroutine-call to the E-5 algorithms will modify the $Q^F\text{-}6(e, R_y)$ image in accordance with the user's insertion or deletion command. (The reason such data-modification operations are only undertaken when y satisfies u(y) is that $Q^F\text{-}6(e, R_y)$ describes only the subset of y-records that satisfy this condition.)

THEOREM 6.4.N. Let e(x, y) denote an E-6 expression and $Q^F\text{-}6(e, R_y)$ denote its data-image. This data-image will support SUM-6, COUNT-6, insertion and deletion algorithms which operate within the standard $\Theta(\log^{D(e)}N)$ runtime. This data-image will also support a FIND-6 algorithm that operates in the standard runtime of $\Theta(l_e(x) + \log^{D(e)}N)$ (which has $l_e(x)$ qualified by the familiar runtime coefficient of 2).

The proof of Theorem 5.5.N is omitted because all aspects of it are trivial consequences of the E-5 theorems and Definitions 6.4.L and 6.4.M.

<u>Remark 6.4.O.</u> Users will frequently desire to modify their list-oriented unary predicates by either adding or deleting a y-record from their unary lists. Such changes will require that the $Q^F\text{-}6(e, R_y)$ data-image be updated to reflect those changes which the user has made in his unary lists. These update operations are fairly easy to execute. They will merely require that the E-6 data-modification algorithm be used to update the $Q^F\text{-}6(e, R_y)$ data-image. Note that Theorem 5.5.N implies that these update operations can be performed in $\Theta(\log^{D(e)}N)$ runtime.

The final topic of this section will be the E-7 expressions. The definition of these expressions was given in Chapter 1. That section indicated that any predicate expression built out of equality, order, and unary atomic predicates is an E-7 expression.

The database algorithms that process E-7 expressions are a fairly straightforward generalization of the E-6 algorithms. These algorithms will take advantage of the fact that every E-7 expression can be decomposed

into a "disjunction sequence" of E-6 expressions. This concept of an E-6 disjunction sequence will be defined in the next paragraph. That concept is important because it will enable the E-7 algorithms to perform their work by making straightforward subroutine-calls to the E-6 algorithms.

Definition 6.4.P. Let $e(x, y)$ denote an E-7 expression and $e_1 e_2 \cdots e_L$ denote a sequence of L distinct E-6 expressions. The $e_1 e_2 \cdots e_L$ predicates will be said to form an E-6 disjunction-sequence of $e(x, y)$ if the following three conditions are met:

1) The $e_1 e_2 \cdots e_L$ predicates must satisfy the disjunction equation given below:

7) $$e(x, y) = \{e_1(x, y) \text{ OR } e_2(x, y) \text{ OR } \cdots e_L(x, y)\} \quad .$$

2) The $e_1 e_2 \cdots e_L$ predicates must furthermore be fully disjoint. (The defintion of "full-disjointness" was given in part 4.4.E of Chapter 4. That definition indicated that $e_1 e_2 \cdots e_L$ are to be judged "fully-disjoint" if and only if no ordered pair can simultaneously satisfy more than one of these predicates.)

3) Each of the $e_1 e_2 \cdots e_L$ predicates must also possess a degree that is less than or equal to that of e. This third condition means that each e_i term of a disjunction-sequence is required to satisfy the $D(e_i) \leq D(e)$ condition.

Comment: Most of the remaining discussion in this section will focus on the E-6 disjunction sequences. That discussion will explain how these disjunction sequences enable the E-7 algorithms to efficiently perform their work by making subroutine-calls to the E-6 algorithms.

Data-Image 6.4.Q. Let $e(x, y)$ denote an E-7 expression, and let $e_1 e_2 \cdots e_L$ denote an E-6 disjunction sequence of this expression. In that case, the union of the L corresponding $Q^F\text{-}6(e_i, R_y)$ data-structures will be said to form an $Q^F\text{-}7(e, R_y)$ data-image.

Observation 6.4.R. It is trivial to verify that all E-7 expressions have an E-6 disjunction sequence. This fact implies that all E-7 expressions will possess a data-image satisfying the previous definition. In some cases, an E-7 expression will possess several distinctly different E-6 disjunction sequences. In these cases, the relevent E-7 expression will possess several distinct $Q^F\text{-}7(e, R_y)$ data-images. Each of these data-images will possess a physically different but functionally equivalent data-structure. In the rest of this section, it will be shown how any one of these $Q^F\text{-}(e, R_y)$ data-images will enable the SUM-7, FIND-7, COUNT-7, and data-modification algorithms to operate in $\log^{D(e)} N$ runtime.

Algorithm 6.4.S. Let $e(x, y)$ denote an E-7 expression whose E-6 disjunction sequence is $e_1 e_2 \cdots e_L$. Let us assume that the $Q^F\text{-}7(e, R_y)$ data-image of this predicate is physically constructed. In this case, a four-

module procedure will be used to perform the SUM, COUNT, FIND and data-modification operations which are associated with e's data-image. These four modules are described below:

1) The SUM-7 module of the present algorithm will be activated whenever the user asks for an $S_e^F(\bar{x})$ value to be calculated. The SUM-7 module will utilize e's disjunction sequence for the purposes of performing this operation. It will calculate the value of $S_e^F(\bar{x})$ by setting it equal to the sum of the sequences' $S_{e_i}^F(\bar{x})$ quantities. More specifically, the SUM-7 module will use a two-step procedure for performing its calculations. The first step will make one subroutine-call to the SUM-6 algorithm for each member of e's disjunction sequence. These subroutine calls will take advantage of the fact that $Q^F\text{-}7(e, R_y)$ is defined to be a union of $Q^F\text{-}6(e_i, R_y)$ substructures. These subroutine-calls will instruct the SUM-6 algorithm to scan each of the $Q^F\text{-}6(e_i, R_y)$ substructures for the purposes of calculating the $S_{e_i}^F(\bar{x})$ values. The second step of the SUM-7 module will calculate the value of $S_e^F(\bar{x})$ by setting it equal to the sum of the previous $S_{e_i}^F(\bar{x})$ quantities.

2) The COUNT-7 module of the present algorithm will calculate the user's $I_e(\bar{x})$'s quantities by using a two-step procedure

that is almost identical to that used by the SUM-7 module. The first step of this procedure will thus instruct the COUNT-6 algorithm to calculate the $I_{e_i}(\bar{x})$ values for each member of e's disjunction sequence. The second step of the COUNT-7 algorithm will calculate the value of $I_e(\bar{x})$ by setting it equal to the sum of these $I_{e_i}(\bar{x})$ quantities.

3) The FIND-7 module of the present algorithm will be activated whenever the user asks for a $Y_e(\bar{x})$ set to be produced. This module will use a procedure that is very similar to the SUM-7 and COUNT-7 modules. The first step of FIND-7 will thus instruct the FIND-6 algorithm to produce each of the disjunction sequence's $Y_{e_i}(\bar{x})$ sets. The second step will set $Y_e(\bar{x})$ equal to the union of the $Y_{e_i}(\bar{x})$ sets.

4) The data-modification module of the present algorithm will execute the user's insertion and deletion commands by making subroutine-calls to the B-6 algorithms and by instructing those procedures to modify each of the $Q^F\text{-}6(e_i, R_y)$ substructures in the manner suggested by the user's insertion and deletion commands.

Example 6.4.T. Let $e(x, y)$ and e_1 e_2 e_3 denote the four predicates defined below:

8) $e(x, y) = \{x.a_1 < y.b_1 \text{ OR } x.a_2 < y.b_2\}$

9) $e_1(x, y) = \{x.a_1 < y.b_1 \text{ AND } x.a_2 < y.b_2\}$

10) $e_2(x, y) = \{x.a_1 \geq y.b_1 \text{ AND } x.a_2 < y.b_2\}$

11) $e_3(x, y) = \{x.a_1 < y.b_1 \text{ AND } x.a_2 \geq y.b_2\}$.

It is easy to verify that the $e_1 e_2 e_3$ predicates jointly constitute an E-6 disjunction sequence for predicate $e(x, y)$. Consequently, the union of the $Q^F\text{-}6(e_1, R_y)$, $Q^F\text{-}6(e_2, R_y)$ and $Q^F\text{-}6(e_3, R_y)$ data-images will constitute an $Q^F\text{-}7(e, R_y)$ data-image. Algorithm 6.4.S will use this data-image during its SUM, COUNT and FIND calculations. Let us watch this algorithm calculate an $S_e^F(x)$ sum value. A two part procedure will be used for such purposes. The first step of SUM-7 will consist of three subroutine-calls to the SUM-6 algorithm. The purpose of these subroutine-calls will be to calculate the $S_{e_1}^F(\bar{x})$, $S_{e_2}^F(\bar{x})$ and $S_{e_3}^F(\bar{x})$ quantities. The second step of the SUM-7 algorithm will calculate the value of $S_e^F(x)$ by setting it equal to the sum of the preceeding three quantities.

<u>THEOREM 6.4.U.</u> Let $e(x, y)$ denote any arbitrary E-7 predicate. For each such E-7 expression

I) There will exist a corresponding $Q^F\text{-}7(e, R_y)$ data-image that describes this predicate,

And this $Q^F\text{-}7(e, R_y)$ data-image will allow

II) The SUM-7 retrieval module to produce $S_e^F(\bar{x})$ sums in $\Theta(\log^{D(e)} N)$ worst-case hashtime.

III) The COUNT-7 module to produce $1_e(\bar{x})$ quantities in $\Theta(\log^{D(e)} N)$ worst-case hashtime.

IV) The FIND-7 module to produce $Y_e(\bar{x})$ set in $\Theta(1_e(x) + \log^{D(e)} N)$ worst-case hashtime (where $1_e(x)$ is qualified by the standard runtime coefficient of 2).

V) The Data-Modification Module to insert or delete any y-record from this data-image in $\Theta(\log^{D(e)} N)$ worst-case hashtime.

VI) And the $Q^F\text{-}7(e, R_y)$ data-image will occupy no more than $\Theta(N \log^{D(e)} N)$ space in computer memory.

<u>Proof of Proposition I:</u> It is very easy to see that every E-7 expression will have an E-6 disjunction sequence (since this is apparent when the E-7 expression is rewritten in disjunction normalized form). This fact together with Definition 6.4.Q implies that every E-7 expression will possess at least one $Q^F\text{-}7(e, R_y)$ data-image.

Q. E. D.

<u>Proof of Proposition II:</u> The proof will be divided into two parts where the correctness and runtime of SUM-7 will be separately verified. Our first goal will be to confirm the correctness of this module.

Note the SUM-7 algorithm was a straightforward procedure based on the following two assumptions

A) If $e_1 e_2 \cdots e_L$ denote the L distinct E-6 predicates which are associated with the $Q^F\text{-}7(e, R_y)$ data-image then this data-image will enable the SUM-6 algorithm to calculate each of

these $S^F_{e_i}(\bar{x})$ values;

B) The sum of the above L distinct $S^F_{e_i}(\bar{x})$ values will equal $S^F_e(\bar{x})$.

Both of the preceeding assumptions are trivial to verify. The first assumption is a consequence of Theorem 6.4.N and the definition of Q^F-7(e, R_y) data-image. The second assertion is a consequence of the definition of E-6 disjunction sequences. The correctness of the SUM-7 retrieval module is an immediate consequence of these two assertions.

It is equally easy to verify the runtime efficiency of the SUM-7 module. Note that if L denotes the number of terms in e's disjunction sequence then the SUM-7 algorithm will make L subroutine-calls to the SUM-6 algorithm. Theorem 6.4.N implies that these L subroutine-calls will consume $\Theta(L \log^{D(e)} N)$ runtime. The preceeding quantity can be regarded to have an $\Theta(\log^{D(e)} N)$ magnitude (because the value of L is unrelated to the size of the R_y relation). Proposition I has thus been proven since the SUM-7 algorithm has been shown to produce its $S^F_e(\bar{x})$ sums in the required runtime.

Q. E. D.

Proofs of Propositions III and IV. Note that the COUNT-7 and FIND-7 algorithms are virtually identical to the SUM-7 algorithm. It thus follows

that the proof of Proposition II can be easily modified so that it also confirms Propositions III and IV.

Q. E. D.

Proof of Proposition V. Let us assume that there are L terms in e's disjunction sequence. Under these circumstances, the E-7 data-modification algorithm will execute the user's insertion or deletion commands by instructing the E-6 algorithm to appropriately update the L corresponding Q^F-6(e_i, R_y) data-image. Note that Theorem 6.4.N stated that each invocation of the E-6 data-modification algorithm will consume no more than $\Theta(\log^{D(e)} N)$ time. All L invocations will thus consume $\Theta(L \log^{D(e)} N)$ time. Note that L is a runtime coefficient (since its value is independent of the size of the database). It thus follows that the E-7 data-modification algorithms must consume no more than $\Theta(\log^{D(e)} N)$ hashtime.

Q. E. D.

Proof of Proposition VI. The techniques that were used to prove Corollary 6.2.F can also be used to confirm Proposition VI. These techniques are applicable because Q^F-7(e, R_y) possesses an $\Theta(\log^{D(e)} N)$ insertion time. They will imply that this data-image can be stored in $\Theta(N \log^{D(e)} N)$ space.

Q. E. D.

Remark 6.4.V. Note that Theorem 5.5.U stated that the E-7 algorithms would operate in a worst-case hashtime that had a $\log^{D(e)} N$ magnitude. This estimate of the expended runtime was quite conservative. In many instances, the E-7 algorithm will operate in a lesser amount of runtime. An example

of such an application can be given if equations 12 through 14 are considered. Note that the latter two equations constitute an E-6 decomposition of equation 12:

12) $e(x,y) = \{[x.a_1 > y.b_1 \text{ AND } x.a_2 > y.b_2] \text{ OR } [x.a_1 < y.b_1 \text{ AND } x.a_3 < y.b_3]\}$

13) $e_1(x,y) = \{x.a_1 > y.b_1 \text{ AND } x.a_2 > y.b_2\}$

14) $e_2(x,y) = \{x.a_1 < y.b_1 \text{ AND } x.a_3 < y.b_3\}$.

The significant aspect about these predicates is that equations 13 and 14 will have $D(e_i)$ values equal to 2 despite the fact that they are the E-6 decomposition of an equation that has a $D(e)$ value equal to 3. The lower $D(e_i)$ values of equations 13 and 14 will insure that the SUM, COUNT, FIND, and data-modification algorithms can process equation 12 in $\log^2 N$ hashtime. This example can be generalized. It will always be possible to possess an E-7 predicate in less than $\log^{D(e)} N$ when it possesses an E-6 decomposition whose constituents have $D(e_i)$ values less than $D(e)$.

Remark 6.4.W. Note that many E-7 predicates will possess several different possible E-6 decompositions. Efficiency will be maximized if our database algorithms will use the best possible E-6 decomposition during their processing of E-7 expressions. Such a judicious choice will enable the E-7 algorithms to improve their runtime coefficient (and sometimes the runtime magnitude will also be improved). A detailed discussion of this topic was

omitted from this thesis for the sake of brevity.

Remark 6.4.X. Note that Definition 6.4.P indicated that all the e_i terms in an E-6 decomposition must be "fully-disjoint." This full disjointness condition was inserted into our definition because it was needed by the SUM-7 and COUNT-7 retrieval algorithms. The significant aspect of this full disjointness condition is that it is not needed by the FIND-7 algorithm. Often, the FIND-7 algorithm will improve its runtime coefficient and magnitude if it builds an E-7 database by taking the union of those data-images which are generated by a nondisjoint set of E-6 predicates. This type of optimization is especially desirable to implement in those applications where $I_e(x)$ has a small value.

Remark 6.4.Y. There are many additional optimizations which can be employed by the E-7 algorithms. All the optimizations which were discussed in the earlier parts of this thesis are obviously applicable to E-7 expressions. Also, an additional important optimization technique is discussed in part 7.3 of this thesis. That section will discuss optimizations that eliminate repeating data-structure from the Q^F-7(e, R_y) data-images. Such redundancies exist in the Q^F-7(e, R_y) data-image because this data-image is begotten by taking the union of several Q^F-6(e_i, R_y) data-images. Section 7.3 will explain how the Q^F-6(e_i, R_y) components of e's disjunction sequence may contain

repeating data-structures. It will also illustrate several optimizing algorithms *that remove these redundancies.*

Observation 6.4.Z. Note that if the results from Theorem 6.4.U were transposed into the two-component runtime terminology (of Section 1.2) then this theorem would state that

1) SUM-7 and COUNT-7 operation can be performed in $\Theta(\log^{D(e)}N; \log^{D(e)}N)$ *worst-case* hashtime.

ii) FIND-7 operations can be performed in $\Theta(\log^{D(e)}N + I_e(x); \log^{D(e)}N)$ worst-case hashtime.

Let us assume that function F maps the elements of R_y into a multiplicative field. This remark will explain how it is possible to generalize result i) in order to develop a MULTIPLY algorithm (for algebraic fields) that has an $\Theta(\log^{D(e)}N; \log^{D(e)}N)$ algorithm when it calculates the product of the $F(y)$ values of the subset of R_y *that satisfies* $e(\bar{x}, y)$. This MULTIPLY algorithm will be mostly a *straightforward generalization* of SUM algorithm (because the latter algorithm has been designed to operate in *any* abelian group). The only aspect of the MULTIPLY algorithm that *differs* from the SUM algorithm can be understood if it is recalled that the presence of element zero will cause multiplication in an arbitrary field to form a semi-group (rather than group). The MULTIPLY algorithm will have to take special measures to process this element of zero. The two main steps of the MULTIPLY algorithm

are listed below:

1) Let Δ denote a field, and let us assume that function F maps R_y into Δ. Also, let $\Delta(\epsilon)$ and $\Delta[\epsilon]$ denote the respective ring of polynomials and field of memorphic functions that have coefficients in Δ and a variable of ϵ. Let G denote a function that maps every non-zero element in Δ onto the same element in $\Delta[\epsilon]$ and which maps element of zero onto the polynomial of ϵ. Let F^{ϕ} denote the composite function of $G \circ F$. The first step of the MULTIPLY algorithm will take advantage of the fact that F^{ϕ} maps R_y into the multiplicative group of non-zero memorphic functions. Note that the SUM algorithm was designed to operate in any abelian group, and it will therefore be capable of calculating the multiplicative product of the $F^{\phi}(y)$ values of the subset of R_y that satisfies $e(\bar{x}, y)$. The first step of our MULTIPLY algorithm will instruct the SUM algorithm to calculate this multiplicative product.

2) Let $p(\epsilon)$ denote the polynomial which was derived by the previous step when it calculated the product of the specified $F^{\phi}(y)$ values. This step will calculate the value which $p(\epsilon)$ will assume when variable ϵ is replaced by constant zero. The MULTIPLY algorithm will inform the user that this

number equals the multiplicative product that he had requested.

It is trivial to verify that the above algorithm will correctly perform multiplication in fields and that it will operate in $\Theta(\log^{D(e)}N; \log^{D(e)}N)$ worst-case hashtime.

6.5 Existential Quantifiers, Universal Quantifier, and Mean Value Algorithms

The algorithms from the last section can be applied to a very diverse set of problems. The next three sections will explain how these procedures can efficiently solve existential quantifier conditions, universal quantifier conditions, mean values, median values, minimum values, and maximum values.

The algorithms for solving these six problems are basically very similar to each other. Each problem will be solved with the aid of the last section's COUNT algorithm. The role of the COUNT algorithm will be to gather statistical information describing the distribution of the data. This statistical information is important because it will enable us to solve these six problems with much greater efficiency than would otherwise be possible.

The first topic will be the existential quantifier problem. The intuitive idea behind the existential quantifier algorithm is very simple. Note that $\bar{x}$ will satisfy the "$\exists_y\, e(\bar{x}, y)$" condition if and only if $I_e(x) \geq 1$. It thus follows that a simple two-step procedure can be used to evaluate existential quantifiers. The first step of this procedure will consist of a subroutine-call to the COUNT algorithm. That subroutine-call will instruct the COUNT algorithm to scan the $Q^F(e, R_y)$ data-image for the purposes of calculating the value of $I_e(\bar{x})$. The second step of the existential-quantifier algorithm will test to see whether the "$I_e(\bar{x}) \geq 1$" condition is satisfied. Obviously, the existential quantifier procedure will report that $\bar{x}$ satisfies $\exists_y\, e(\bar{x}, y)$

when the preceeding condition is met.

A basically similar procedure will be used to evaluate universal quantifiers. During the discussion of this topic, N will denote the size of the R_y relation. The universal quantifier algorithm will take advantage of the fact that $\bar{x}$ satisfies "$\forall_y\, e(x, y)$" if and only if $I_e(\bar{x}) = N$. It will use a two-step procedure for evaluating universal quantifier. The first step of this procedure will instruct the COUNT algorithm to calculate the value of $I_e(\bar{x})$. The second step of our algorithm will check to see whether the "$I_e(\bar{x}) = N$" condition is satisfied. The user will be told that $\bar{x}$ satisfies the universal quantifier when the preceeding condition holds.

A very simple procedure will be also used to calculate the mean $F(y)$ value for those y-records that satisfy $e(\bar{x}, y)$. This procedure will have three steps. The first two steps will consist of subroutine-calls to the SUM and COUNT algorithms. The invoked algorithms will be instructed to search the $Q^F(e, R_y)$ data-image for the purposes of calculating the values of $S_e^F(\bar{x})$ and $I_e(\bar{x})$. The third step of our mean-value procedure will calculate the quotient of these two numbers. That quotient will obviously represent the desired mean-value.

THEOREM 6.5.A. Let $e(x, y)$ denote an E-7 expression, and let us assume that its $Q^F(e, R_y)$ data-image is physically constructed. The algorithms described in the previous three paragraphs will then

I) Correctly evaluate universal quantifiers, existential quantifiers, and mean values;

II) And perform these three calculation in $\Theta(\log^{D(e)} N)$ worst-case hash-runtime.

Proof of Proposition I. Note that the existential quantifier, universal quantifier, and mean-value algorithms were straightforward extenciona of the last section's SUM-7 and COUNT-7 algorithm. Also note that Theorem 6.4.U asserted the correctness of the SUM-7 and COUNT-7 procedures. The correctness of the existential quantifier, universal quantifier and mean-value algorithms thus follows from this proposition. Q. E. D.

Proof of Proposition II. It is very easy to analyze the runtime of the existential-quantifier, universal-quantifier and mean-value procedures. The only time-consuming part of the first two procedures was their subroutine-calls to the COUNT-7 algorithm. Theorem 6.4.U indicates that these subroutine-calls will take $\Theta(\log^{D(e)} N)$ time. The runtime of the mean-value algorithm can be analyzed once it is noted that the two time-consuming parts of this procedure are its subroutine-calls to the SUM-7 and COUNT-7 algorithms. Theorem 6.4.U indicates that these steps will consume $\Theta(\log^{D(e)} N)$ time. The proof has thus been completed since we have confirmed that the universal-quantifier, existential quantifier, and mean-value algorithms will operate in $\Theta(\log^{D(e)} N)$ worst-case runtime. Q. E. D.

Observation 6.5.B. The combined results of Theorems 6.4.U and 6.5.A clearly illustrate the importance of the $Q^F\text{-}\mathcal{T}(e, R_y)$ data-image. Note that these theorems have shown that $Q^F\text{-}\mathcal{T}(e, R_y)$ supports

I) SUM, COUNT, mean-value, universal quantifier and existential quantifier queries that can be performed in $\Theta(\log^{D(e)} N)$ worst-case hashtime

II) FIND operations which produce their $Y_e(x)$ sets in $\Theta(I_e(\bar{x}) + \log^{D(e)} N)$ worst-case hashtime

III) data-modification operations which can insert or delete a y-record in $\Theta(\log^{D(e)} N)$ worst-case hashtime

IV) and the previous theorems have also demonstrated that the $Q^F\text{-}\mathcal{T}(e, R_y)$ data-image will consume no more than $\Theta(N \log^{D(e)} N)$ space in computer memory.

Clearly these combined results have illustrated the usefulness of the $Q^F\text{-}\mathcal{T}(e, R_y)$ data-image.

6.6 Multi-variable Generalizations

The discussion in the previous sections were restricted to algorithms whic' processed two-variable $e(x, y)$ predicates. It is fairly easy to generalize these algorithms for the cases of the multi-variable predicates. This section will discuss such multi-variable generalizations. This subject is important both because many users will desire to employ such procedures and because the next section will show how these algorithms can be applied to solving minimum-value, maximum-value and median-value problems.

The general nature of the multi-variable problem can be best illustrated if the example of the SUM algorithm is considered. In the forthcoming discussion, $e(x_1 x_2 \cdots x_k, y)$ will denote a $(k+1)$-variabled E-T predicate expression, and $S_e^F(\bar{x}_1 \bar{x}_2 \cdots \bar{x}_k)$ will denote the sum of the $F(y)$ values of those y-records which satisfy $e(\bar{x}_1 \bar{x}_2 \cdots \bar{x}_k, y)$.

The algorithm that calculates multi-variable sums is an extremely simple extension of the two-variable SUM algorithm. The distinction between these algorithms is largely one of terminology. In the discussion which follows, we shall assume that our $x_1 x_2 \cdots x_k$ variables are each j-tuples and that x^* denotes that kj-tuple begotten by merging these k distinct j-tuple into a single compound variable. Clearly this terminology implies that $e(x_1 x_2 \cdots x_k, y)$ and $e(x^*, y)$ are logically equivalent predicates (since x^* represents the variable produced when merging the $x_1 x_2 \cdots x_k$ variables

into a single term). Note that section 6.4's SUM-7 algorithm is capable of calculating the value of $S_e^P(\bar{x}^*)$. It thus follows that the multi-variable sum algorithm can calculate the value of $S_e^P(\bar{x}_1 \bar{x}_2 \cdots \bar{x}_k)$ by simply setting it equal to the preceeding value which SUM-7 can calculate.

It is equally easy to produce multi-variable algorithms that perform FIND, COUNT, mean-value, existential-quantifier and universal-quantifier tasks. Each of these algorithms will presuppose that the user supplies the algorithm with an $e(x_1 x_2 \cdots x_k, y)$ predicate and k arguments of $\bar{x}_1 \bar{x}_2 \cdots \bar{x}_k$. All of the multi-variable algorithms will perform their tasks by making similar subroutine-call to the two-variable algorithms.

The multi-variable algorithms were briefly discussed in this chapter for two reasons. The first reason is that many users will desire to perform such operations. The multi-variable algorithms are also important because they will be needed by the next chapter's minimum-value, median-value, and maximum-value procedures.

6.7 The Maximum, Minimum, and Median-Value Algorithms

Techniques that can be used to locate the maximum, minimum, median, and general i-th smallest members of that subset of y-records that satisfy $e(\bar{x}, y)$ will be discussed in this section. The goal of this section will be to explain how a series of judicious subroutine-calls to Section 6.4's COUNT algorithm will produce search algorithms that perform these tasks in $\Theta(\log^{D(e)+2} N; \log^{D(e)+1} N)$ worst-case hashtime.

The discussion in this section will be divided into three parts. The first two parts will introduce two initial algorithms, called COMPARE-COUNT and LOCATE. The third part of this section will explain how these algorithms can be applied to the minimum-value, median-value and maximum-value search tasks.

The COMPARE-COUNT algorithm will be a procedure that requires four arguments. These arguments will be a predicate of $e(x, y)$, a fixed element of $\bar{x}$, a y-attribute of b, and a real number of $\bar{r}$. The purpose of the COMPARE-COUNT algorithm will be to use these four arguments to produce a resulting number of $1_{e_b}(\bar{x}, \bar{r})$. This quantity will be defined to be the COUNT of how many y-records simultaneously satisfy $e(\bar{x}, y)$ and $y.b \leq \bar{r}$.

The COMPARE-COUNT algorithm will produce the preceeding quantity by making a subroutine-call to section 6.4's COUNT algorithm.

The mechanics of this process can be best explained if some additional notation is introduced. Let $e_b(x, y, r)$ denote the predicate given in equation 1, and let $Q^F(e_b, R_y)$ denote the data-image of this predicate

1) $$e_b(x, y, r) = \{e(x, y) \text{ AND } y.b \leq \bar{r}\} \quad .$$

The COMPARE-COUNT algorithm will produce its $I_{e_b}(\bar{x}, \bar{r})$ number by instructing the COUNT algorithm to search the $Q^F(e_b, R_y)$ data-image with the $e_b(\bar{x}, y, \bar{r})$ predicate.

LEMMA 6.7.A. Let $e(x, y)$ denote an E-7 predicate, and let us assume that the $Q^F(e_b, R_y)$ data-image is physically constructed. The COMPARE-COUNT algorithm will then

I) Operate correctly;

II) And perform its retrieval operation in $\Theta(\log^{D(e)+1} N)$ worst-case hashtime.

Proof: The lemma is completely trivial to verify. The correctness of the COMPARE-COUNT algorithm follows from Theorem 6.4.U (which stated that this thesis's COUNT algorithms always correctly processes E-7 expressions). Our analysis of the runtime of the COMPARE-COUNT algorithm will be based on the following two observations:

1) The value of $D(e_b)$ will always equal either $D(e)$ or $D(e) + 1$.

2) Theorem 6.4.U implies that the COUNT algorithm will consume $\Theta(\log^{D(e_b)} N)$ worst-case hash runtime when it is called by the COMPARE-COUNT algorithm.

The preceeding two facts clearly imply that the COMPARE-COUNT algorithm can never consume more than $\Theta(\log^{D(e)+1} N)$ hash-runtime. The lemma is thus proven (since we have verified both of its propositions). Q. E. D.

LEMMA 6.7.B. A y-record can always be inserted into or deleted from the $Q^F(e_b, R_y)$ data-image in $\Theta(\log^{D(e)+1} N)$ worst-case hash-runtime.

Proof: This lemma is also a consequence of Theorem 6.4.U. Note that the cited theorem implied that $Q^F(e_b, R_y)$ supports $\Theta(\log^{D(e_b)} N)$ insertion and deletion operations. Clearly, the preceeding runtime estimate together with the $D(e_b) \leq D(e) + 1$ condition implies that insertion and deletion operations can always be performed in $\Theta(\log^{D(e)+1} N)$ worst-case time. Q. E. D.

Remark 6.7.C. It must be emphasized that the $\Theta(\log^{D(e)+1} N)$ runtimes in the previous two theorems were worst-case estimates. In many cases, the preceeding algorithms will operate in better than their worst-case runtimes. There are two reasons for this. These are that:

1) The proofs in the preceeding theorems conservatively assumed that $D(e_b)$ equaled $D(e) + 1$. In many cases, $D(e_b)$ will be equal to the smaller quantity of $D(e)$. In these cases, the cited

algorithms will operate in the improved $\Theta(\log^{D(e)} N)$ runtimes.

7) Also, note that Remark 6.4.V showed that the COUNT, insertion and deletion algorithms will often operate in runtimes that are considerably better than the previous worst-case estimates. Obviously, the algorithms in this section will have a better runtime in those cases where their subroutine-calls operate with better-than-expected efficiencies.

The next topic of this section will be the LOCATE algorithm. This algorithm will accept four arguments. These arguments will be a predicate of $e(x, y)$, an element of $\bar{x}$, a y-attribute of b, and an integer of i. The purpose of the LOCATE algorithm will be to scan the set of y-records that satisfy $e(\bar{x}, y)$ for the purposes of finding that member of this subset which has the i-th smallest y.b value. This y-record will henceforth be denoted as LOCATE(e, b, $\bar{x}$, i), and it will be called the "target record." The main goal of this section will be to prove that the LOCATE algorithm can retrieve the target record in $\Theta(\log^{D(e)+2} N)$ worst-case hashtime.

During our discussion, we will take advantage of the fact that the $Q^F(e_b, R_y)$ data-image will contain a Super-B-Tree that has arranged the elements of R_y in order by increasing y.b value. Let v denote some node of this Super-B-Tree. Throughout this section, the symbol $\bar{r}_v$ will denote the maximum y.b-value associated with v's left subtree. Note

that it can be presumed that:

1) All the y-leaves of v's left subtree will have $y.b \leq \bar{r}_v$.

2) All the y-leaves of v's right subtree will have $y.b \geq \bar{r}_v$.

3) Our Super-B-Tree will contain a sufficiently large amount of information in node v for $\bar{r}_v$ to be calculated in $\Theta(1)$ time.

The LOCATE algorithm will use the preceeding r_v values to help it perform its y-leaf searches. The algorithm will use a tree walking procedure to help it find the i-th smallest y-leaf that satisfies $e(\bar{x}, y)$. This tree walking procedure will begin at the tree root, and it will move downwards towards the sought-after leaf. During this search, the LOCATE algorithm will pause at each interior node, and it will determine whether the target-leaf is a member of that record's left as opposed to right subtree. This determination will be made via a subroutine-call to the COMPARE-COUNT algorithm. That procedure will be instructed to return the number of $I_{e_b}(\bar{x}, \bar{r}_v)$ to the LOCATE algorithm. The LOCATE algorithm will test this number against the "$i \leq I_{e_b}(\bar{x}, \bar{r}_v)$" condition. The i-th smallest y-record will be determined to be a member of v's left subtree when this condition is met; and it will be judged to be a member of v's right subtree when this condition fails. The LOCATE algorithm will use the preceeding tests to determine which of v's two sons should be examined during its recursive search for the i-th smallest y-leaf. The LOCATE algorithm

will thus move to the appropriate son, and it will continue recursively processing of that record and its descendants until the sought-after target leaf is found.

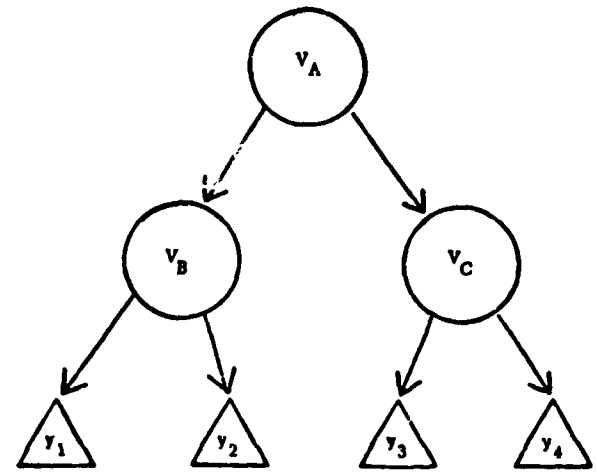
Example 6.4.D. Let us watch the LOCATE algorithm process the following predicate:

2) $e(x, y) = \{x.a_1 = y.b_1\}$.

In this example, we will assume that the R_y relation will possess the four y-record described below:

record-name	$y.b_1$ value	$y.b_2$ value
$\bar{y}_1$	11	21
$\bar{y}_2$	12	22
$\bar{y}_3$	12	23
$\bar{y}_4$	12	24

Note that the $Q^P(e_{b_2}, R_y)$ data-image of this example will possess a tree-structure that lists the elements of R_y in order by increasing $y.b_2$ value. One possible representation of that tree is shown in the diagram below. In that diagram, the interior nodes are represented as circles, and the leaves are represented as triangles. Also, note that r_{v_A}, r_{v_B} and r_{v_C} will have respective values of 22, 21, and 23 in this example.



Let $\bar{x}$ denote an element whose $\bar{x}.a_1$ value equals 12. Note that the $\bar{y}_2$, $\bar{y}_3$, and $\bar{y}_4$ records satisfy the $e(\bar{x}, y)$ condition. In this example, we will watch the LOCATE algorithm find the second smallest y-record that satisfies $e(\bar{x}, y)$.

The LOCATE algorithm will begin at the tree-root of v_A. It will check to see whether this node satisfies the "$2 \leq I_{e_{b_2}}(\bar{x}, \bar{r}_{v_A})$" condition. This condition will fail because $I_{e_{b_2}}(\bar{x}, \bar{r}_{v_A})$ is equal to one. Consequently, the LOCATE algorithm will conclude that the second smallest y-record

must belong to the right subtree of v_A. The LOCATE algorithm will therefore proceed to the node of v_C. The algorithm will conduct a similar test on this node. It will thus check to see whether the "$2 \le I_{e_{b_2}}(\bar{x}, \bar{r}_{v_C})$" condition is satisfied. This condition will pass the preceeding test because $I_{e_{b_2}}(\bar{x}, \bar{r}_{v_C})$ equals 2. Consequently, the LOCATE algorithm will conclude that the second smallest y-element is the left son of v_C. The algorithm will therefore inform the user that $\bar{y}_3$ is the second smallest of the three y-records that satisfy $e(\bar{x}, y)$.

THEOREM 6.4.E. Let us assume that the $Q^F(e_b, R_y)$ data-image is physically constructed and that the LOCATE algorithm is given the arguments of e, b, $\bar{x}$, and i. Under these circumstances, the LOCATE algorithm:

I) Will correctly find that y-record which has the i-th smallest y.b value among the set of y-records that satisfy $e(\bar{x}, y)$;

II) And will perform this task in $\Theta(\log^{D(e)+2} N)$ worst-case hash-time.

Proof of Proposition I. Note that the LOCATE algorithm was a search procedure based on the two assumptions that:

A) The target y-record will have its y.b value less than or equal to r if and only if $i \le I_{e_b}(\bar{x}, \bar{r})$;

B) The COMPARE-COUNT algorithm will correctly calculate the value of $I_{e_b}(\bar{x}, \bar{r})$.

Both of these assumptions are easy to verify. The definition of $I_{e_b}(\bar{x}, \bar{r})$ implies the correctness of the first assumption. Also, Lemma 6.7.A confirms the correctness of the second assumption. An examination of the LOCATE algorithm indicates that the correctness of this algorithm is an immediate consequence of the previous two assertions. Q.E.D.

Proof of Proposition II. Note that the only time-consuming part of the LOCATE algorithm was its subroutine-calls to the COMPARE-COUNT algorithm. The analysis of these subroutine-calls will be based on the following three observations:

A) Lemma 6.7.A indicated that each invocation of the COMPARE-COUNT algorithm will consume $\Theta(\log^{D(e)+1} N)$ worst-case hashtime;

B) There will be no more than h invocations of the COMPARE-COUNT algorithm when its y.b Super-B-Tree has a height equal to h;

C) The preceeding tree will have an $\Theta(\log N)$ height (because it is a B-tree).

Clearly, assumptions B and C indicate that the LOCATE algorithm will make $\Theta(\log N)$ subroutine-calls to the COMPARE-COUNT algorithms. This together with observation A clearly implies that a total of $\Theta(\log^{D(e)+2} N)$ worst-case hashtime will be consumed by these subroutine-calls.

Q. E. D.

The final topic of this section will be the algorithms that retrieve the maximum, median and minimum members of the subset of y-records that satisfy $e(\bar{x}, y)$. A two-step procedure will be used to perform such search operations. These algorithms will begin by instructing the COUNT-7 algorithm to calculate the value of $I_e(\bar{x})$. The three search algorithms will subsequently make subroutine-calls to the LOCATE algorithm for retrieving their target records. Each of these three subroutine-calls will use the standard four arguments of e, b, $\bar{x}$ and i. The only difference between the maximum, median and minimum search operations is that i will be given the three respective values of $I_e(\bar{x})$, $(I_e(x) + 1)/2$, and 1 during the subroutine-calls to the LOCATE algorithm. The three target records returned by the LOCATE algorithm will represent the sought-after maximum, median, and minimum y-records.

THEOREM 6.7.F. Let us assume that the $Q^F(e_b, R_y)$ data-image is physically constructed and that the user has requested that the above algorithms search the set of y-records which satisfy $e(\bar{x}, y)$ for the purposes of

retrieving the members of this set that have maximum, median, and minimum $y.b$ values. The above described algorithms can perform these search operations in $\Theta(\log^{D(e)+2} N)$ worst-case hashtime.

Proof: Note that Theorem 6.4.U showed that the COUNT algorithm operates in $\Theta(\log^{D(e)} N)$ time and that Theorem 6.7.E showed that the LOCATE algorithm operates in $\Theta(\log^{D(e)+2} N)$ retrieval-time. The maximum, median, and minimum algorithms are of course simple extensions of the previous two algorithms. Their correctness and runtime efficiency is thus an immediate consequence of the previous theorems.

Q. E. D.

Observation 6.7.G. Note that Lemma 6.7.B indicated that a y-record can be inserted into or deleted from $Q^F(e_b, R_y)$ in $\Theta(\log^{D(e)+1} N)$ worst-case hashtime. This fact combined with the previous two retrieval theorems implies that the LOCATE, maximum-value, median-value and minimum-value algorithms can operate in $\Theta(\log^{D(e)+2} N ; \log^{D(e)+1} N)$ worst-case hashtime.

Remark 6.7.H. It should be stated that the previous remark and theorems gave very conservative worst-case estimates to the runtime of our various algorithms. This is because the LOCATE algorithm was a procedure that was built around subroutine-calls to the COMPARE-COUNT algorithm. Note that Remark 6.7.C indicated that COMPARE-COUNT often operated far more efficiently than was indicated in our worst-case estimates. Obviously, the same is true of the LOCATE, maximum-value, median-value, and

minimum-value algorithms (since they make subroutine-calls to COMPARE-COUNT).

Remark 6.7.I. It should be stated that the proposed search algorithms were primarily intended for those applications where the $Y_e(\bar{x})$ set has a large size. When this set has a small size, it is usually preferable to retrieve the minimum, median, maximum, and i-th smallest y-record by making an exhaustive search through the $Y_e(\bar{x})$ set. Such a search-by-exhaustion procedure will rely upon the FIND algorithm to construct the needed $Y_e(\bar{x})$ set. The total time consumed by this search method will thus have an $\Theta(I_e(\bar{x}) + \log^{D(e)}N ; \log^{D(e)}N)$ magnitude.

All the results that were proven in this chapter are summarized by Theorem 6.7.J. That theorem will use the two component runtime terminology to describe the previous database algorithms:

THEOREM 6.7.J. Let $e(x, y)$ denote an E-7 expression. It will then follow that:

I) The SUM, COUNT, existential-quantifier, universal-quantifier and mean-value algorithms can process this expression in $\Theta(\log^{D(e)}N ; \log^{D(e)}N)$ worst-case hash-runtimes;

II) The FIND algorithm can produce the $Y_e(x)$ sets for this expression in $\Theta(I_e(x) + \log^{D(e)}N ; \log^{D(e)}N)$ worst-case hashtime

(where $I_e(x)$ is qualified by a runtime coefficient of 2).

III) The minimum, median, maximum, and LOCATE algorithms will process this expression in $\Theta(\log^{D(e)+2}N ; \log^{D(e)+1}N)$ worst-case hash-runtimes.

The proof of Theorem 6.7.J is omitted because this proposition is a summary of results that were previously verified in Theorems 6.4.U, 6.5.A, 6.7.B, 6.7.E and 6.7.F. The reader should note that the first components of the preceeding runtimes represent the amount of time needed to perform retrieval operations, and the second components correspond to the amount of time needed to update the data-image when the user gives an insertion or deletion command. Note that the SUM, COUNT, FIND, mean-value, existential-quantifier, and universal-quantifier algorithms had a $\log^{D(e)}$ data-modification time whereas the minimum-value, median-value, maximum-value and LOCATE algorithm had a $\log^{D(e)+1}N$ data-modification time. This discrepancy is due to the fact that the first six algorithms used the $Q^F(e, R_y)$ data-image and that the last four algorithms instead used the $Q^F(e_b, R_y)$ data-image.

7.1 Chapter Overview

This chapter will discuss several further modules that can be added to the ten basic algorithms that were proposed in this thesis. There will be four sections in this chapter.

Section 7.2 will show how the basic database algorithms of this thesis can be generalized so that they can process a primitive called the tabular predicate. Tabular predicates are important because they will have many useful commercial applications.

The second topic of this chapter will be the special kinds of optimizations which are possible when the user wishes to perform queries on several different predicates. Often, it will be possible to design a compound database that simultaneously serves several predicates. The advantage of such a compound database is that it will often serve the predicates more efficiently than a collection of several distinct data-structures for the individual predicates. Such compound databases will be studied in sections 7.3 and 7.4 of this chapter.

The final part of this chapter will be section 7.5. That section will discuss several additional theorems whose proofs were omitted from this thesis for the sake of brevity.

It must be emphasized that all the concepts in this chapter are extremely important. They will collectively illustrate how the techniques of this thesis can be applied to virtually all classes of commercial user requests.

7.2 Tabular Predicates and the E-8 Algorithms

Let R_x and R_y denote two relations. A "subset-table for the R_xR_y crossproduct" will be defined to be a list of ordered pairs whose first components are pointers to elements of R_x and whose second components are pointers to elements of R_y. Throughout this thesis, the symbols of $\sigma_1^{xy}\ \sigma_2^{xy} \ldots \sigma_k^{xy}$ will denote a collection of such subset-tables.

Definition 7.2.A. Let σ_i^{xy} denote a subset-table of the R_xR_y cross-product. Each such subset-table will be said to possess an associated "tabular predicate." The symbol of $T_i(x,y)$ will denote this tabular predicate. $T_i(x,y)$ will be defined to be that predicate which assigns a boolean value of "TRUE" to precisely those xy-ordered pairs which appear in the σ_i^{xy} subset-table.

Example 7.2.B. Suppose that σ_1^{xy} subset-table contains the three ordered pairs shown below:

first-component	second-component
pointer to $\bar{x}_1$	pointer to $\bar{y}_1$
pointer to $\bar{x}_2$	pointer to $\bar{y}_1$
pointer to $\bar{x}_2$	pointer to $\bar{y}_2$

Under these circumstances, the set of ordered pairs satisfying $T_1(x,y)$ will be $(\bar{x}_1,\bar{y}_1)$, $(\bar{x}_2,\bar{y}_1)$ and $(\bar{x}_2,\bar{y}_2)$.

The tabular-predicates are very important because they will allow the user to describe many logical concepts that would otherwise be awkward to formalize. The advantage of these tabular predicates can be seen if the example is considered where a user wishes to formalize the concept that a given product is sold by a given store. Such a relationship could be described by a subset-table whose first components are pointers to individual stores and whose second components are pointers to the items which are sold in the specified stores. The advantage to such a subset-table and its tabular predicate is that they allow the user to express a concept which would otherwise be awkward to formulate.

The purpose of this section will be to generalize the previous database algorithms so that they can also process tabular predicates. During that discussion, we will speak of the E-8 class of predicate expressions. Intuitively, this class will represent the natural tabular predicate generalization of the "E-7 expressions." A more formal definition of E-8 expressions is given below.

Definition 7.2.C. A predicate, $e(x,y)$, will be said to be an E-8 expression if it is built out of atomic tabular, equality, order and unary predicates.

A slight amount of additional notation must be introduced before the statement of the main E-8 theorems can be given. During our discussion, the symbol of $\mathcal{T}_i(\bar{x})$ will denote the set of y-values

associated with those ordered pairs in the $\mathcal{T}_i^{xy}$ table which have a first component equal to $\bar{x}$. The symbol of $N_i(\bar{x})$ will denote the number of y-records that belong to this $\mathcal{T}_i(\bar{x})$ set. Finally, the symbol of $N^e(\bar{x})$ will denote the sum of the $N_i(\bar{x})$ values of those $T_i(x,y)$ tabular predicates that appear in the $e(x,y)$ predicate.

The main theorem of this section will use the preceding $N^e(\bar{x})$ quantity to describe the runtime of the various E-8 algorithms. That proposition will state that the difference in the amount of time required by the E-7 and E-8 algorithms will be fairly small. This theorem will state that this difference is equal to $N^e(\bar{x})$ time for the cases of the SUM, FIND, COUNT, universal-quantifier, existential-quantifier, and mean-value algorithms, and that it is equal to $N^e(\bar{x}) \cdot \log N$ time for the cases of the LOCATE, minimum-value, median-value, and maximum-value algorithms.

The rest of this section will be devoted to describing and proving the correctness of the various E-8 theorems. It must be stated that the E-8 algorithms will be very trivial extensions of the E-7 algorithms. The importance of the E-8 algorithms will result from the fact that most commercial users will prefer to express their queries in terms of E-8 rather than E-7 expressions. The discussion in this section will thus be important from a practical, but not matnematical point of view. Readers who are only interested in the mathematical aspects of this thesis should probably skim the remainder of this section.

Data-Structure 7.2.D. During our discussion, it will be assumed that the $\mathcal{T}_i^{xy}$ subset-tables are given a data-format with the following two characteristics:

1) These tables should possess "inverted file" data-structures. The significant characteristic of these inverted files should be that they enable the user to find the $N_i(\bar{x})$ members of any corresponding $\mathcal{T}_i(\bar{x})$ subset in $N_i(\bar{x})$ time.

2) The $\mathcal{T}_i^{xy}$ table should also possess a flexible data-format that allows the user to insert or delete a given ordered pair in $\mathcal{O}(1)$ hashtime.

It is trivial to verify that data-structures which satisfy the previous two conditions can be designed by combining well-known inverted file techniques with section 3.2's hashing functions.

The database algorithms in this section will process E-8 expressions by using intermediate terms which are called "reduction predicates." The definition of these reduction predicates is given below:

Definition 7.2.E. Let e(x,y) denote an E-8 expression and let us assume that the symbols of $T_1(x,y)$, $T_2(x,y)$ ... $T_K(x,y)$ denote the set of tabular predicates which appear within expression e. Let i denote an integer whose value is between zero and K. Throughout this section, the symbol of $e_i^{\pi}(x,y)$ will denote that predicate which is

identical to e(x,y) except that each occurrence of a $T_1(x,y)$, $T_2(x,y)$ ... $T_i(x,y)$ tabular predicate in e(x,y) is replaced by a boolean value of FALSE in $e_i^{\pi}(x,y)$.

Example 7.2.F. Let us calculate the e_0^{π}, e_1^{π}, and e_2^{π} reduction predicates for the following E-8 expression:

(1) $e(x,y) = \{[x.a_1 = y.b_1 \text{ AND } T_1(x,y)] \text{ OR } [x.a_2 = y.b_2 \text{ AND NOT } T_2(x,y)]\}$

The reduction predicates that equation 1 produces are shown in equations 2 through 4. Note that the $e_0^{\pi}(x,y)$ reduction predicate is equivalent to e(x,y). This is because there are no tabular predicates removed during the "zero-th reduction":

(2) $e_2^{\pi}(x,y) = \{[x.a_1 = y.b_1 \text{ AND FALSE}] \text{ OR } [x.a_2 = y.b_2 \text{ AND NOT FALSE}]\}$

(3) $e_1^{\pi}(x,y) = \{[x.a_1 = y.b_1 \text{ AND FALSE}] \text{ OR } [x.a_2 = y.b_2 \text{ AND NOT } T_2(x,y)]\}$

(4) $e_0^{\pi}(x,y) = e(x,y)$

Throughout this section, we will always try to simplify our reduction predicates by transforming them into logically equivalent expressions that do not contain the boolean constant of FALSE. The simplified versions of the e_1^{π} and e_2^{π} reductions are shown in equations 5 and 6. Note that the e_1^{π} and e_2^{π} expressions in these equations represent equivalent but simpler versions of the predicates which were shown in equations 2 and 3.

(5) $e_2^{\pi}(x,y) = \{x.a_2 = y.b_2\}$

(6) $e_1^{\pi}(x,y) = \{x.a_2 = y.b_2 \text{ AND NOT } T_2(x,y)\}$

Observation 7.2.G. *Throughout this section, the symbol of K will be used to denote the number of atomic predicates which appear in* $e(x,y)$. Note that our notation implies that all K of e's tabular predicates are removed in the $e_K^{\pi}(x,y)$ reduction predicate. This fact is very important. It implies that $e_K^{\pi}(x,y)$ is an E-7 expression. The algorithms discussed in the later parts of this section will take advantage of this fact. They will make subroutine-calls to the E-7 algorithms and instruct the invoked procedures to process predicate $e_K^{\pi}(x,y)$.

Observation 7.2.H. Note that Definition 7.2.E implies that the $e_0^{\pi}(x,y)$ predicate is equivalent to $e(x,y)$ (because no tabular predicates are removed during the "zero-th reduction"). This fact will also be used by most of the theorems in this section.

Definition 7.2.I. Let $S^F_{e_i^{\pi}}(x)$ denote the sum associated with the $e_i^{\pi}(x,y)$ predicate. During this section, the symbol $Z_i^F(x)$ will be an abbreviation for the following arithmetic expression:

(7) $Z_i^F(x) = S^F_{e_{i-1}^{\pi}}(x) - S^F_{e_i^{\pi}}(x)$

Data-Image 7.2.J. Let $e(x,y)$ denote an E-8 expression which possesses K tabular predicate, and let $e_K^{\pi}(x,y)$ denote the corresponding K-th reduction of this predicate. The symbol of $Q^F\text{-}8(e,R_y)$ will henceforth denote the data-image that is associated with predicate $e(x,y)$. This data-image will be defined to possess a data-structure which is identical to that of $Q^F\text{-}7(e_K^{\pi},R_y)$. (The latter data-image can be presumed to be well defined because Observation 7.2.G showed that $e_K^{\pi}(x,y)$ was an E-7 expression.)

The rest of this section will explain how the $Q^F\text{-}8(e,R_y)$ data-image enables our various database algorithms to efficiently process E-8 expressions. This discussion will begin with a detailed description of the SUM-8 retrieval algorithm. That discussion will be very useful because the other E-8 algorithms will be very similar to SUM-8.

Algorithm 7.2.K. Let $e(x,y)$ denote an E-8 expression, and let us assume that the $Q^F\text{-}8(e,R_y)$ data-image is physically constructed. The SUM-8 retrieval algorithm will be designed to scan this data-image for the purposes of calculating $S_e^F(\bar{x})$. This algorithm will execute the following three steps to calculate $S_e^F(\bar{x})$:

1) The first step will make a subroutine-call to SUM-7 and instruct that algorithm to search the $Q^F\text{-}8(e,R_y)$ data-image with the $e_K^{\pi}(\bar{x},y)$ predicate. The purpose of this search will be to produce the subtotal of $S^F_{e_K^{\pi}}(\bar{x})$.

2) Let us recall that the definition of $Z_1^F(\bar{x}), Z_2^F(\bar{x}) \ldots Z_K^F(\bar{x})$ was given in 7.2.1. The second step of the SUM-8 algorithm will be designed to calculate the values of $Z_1^F(\bar{x}), Z_2^F(\bar{x}) \ldots Z_K^F(\bar{x})$. Each $Z_i^F(\bar{x})$ quantity will be calculated by a similar procedure. That procedure will walk the $\mathcal{F}_i(\bar{x})$ set and calculate the $e_i^\pi(\bar{x},\bar{y})$ and $e_{i-1}^\pi(\bar{x},\bar{y})$ truth-values for each y-member of this set. Let us use the integers of one and zero to denote the respective boolean values of TRUE and FALSE. This step will calculate the value of $Z_i^F(\bar{x})$ by setting it equal to the sum of the $\{[e_{i-1}^\pi(\bar{x},\bar{y}) - e_i^\pi(\bar{x},\bar{y})] \cdot F(\bar{y})\}$ values of those y-records that belong to $\mathcal{F}_i(\bar{x})$.

3) The third step will calculate the value of $S_e^F(\bar{x})$ by using the following formula:

(8) $$S_e^F(\bar{x}) = S^F_{e_K^\pi}(\bar{x}) + \sum_{i=1}^{K} Z_i^K(\bar{x})$$

<u>Example 7.2.L.</u> Let $T_1(x,y)$ and $T_2(x,y)$ denote two tabular predicates. In this example, we will watch the SUM-8 algorithm process the E-8 predicate that is defined below:

(9) $e(x,y) = \{T_1(x,y) \text{ OR } \{x.a = y.b \text{ AND NOT } T_2(x,y)\}\}$

During our discussion, it will be assumed that the following table

describes the R_y relation:

element-name	y.b-value	F(y)-value
$\bar{y}_1$	1	101
$\bar{y}_2$	1	102
$\bar{y}_3$	1	103
$\bar{y}_4$	1	104
$\bar{y}_5$	2	105
$\bar{y}_6$	2	106

The SUM-8 algorithm will use the reduction predicates of e_0^π, e_1^π and e_2^π during its calculation of $S_e^F(\bar{x})$. The general nature of reduction-predicates was explained in Definition 7.2.E. The reduction predicates that are used by the present example are listed below:

(10) $e_2^\pi(x,y) = \{x.a = y.b\}$

(11) $e_1^\pi(x,y) = \{x.a = y.b \text{ AND NOT } T_2(x,y)\}$

(12) $e_0^\pi(x,y) = e(x,y)$

During this example, it will be assumed that the SUM-8 algorithm is processing an x-element which satisfies the following three conditions:

A) The value of $\bar{x}.a$ equals 1;

B) The records of $\bar{y}_2\bar{y}_3\bar{y}_6$ constitute the set of elements that satisfy $T_1(\bar{x},y)$;

C) The records of $\bar{y}_3\bar{y}_4\bar{y}_5$ constitute the set of y-records that satisfy $T_2(\bar{x},y)$.

The SUM-8 procedure will calculate the value of $S_e^F(\bar{x})$ for such an x-element in the manner suggested by Algorithm 7.2.K. The details of this procedure are described in the next several paragraphs.

The first step of the SUM-8 algorithm will calculate the value of $S^F_{e_2^\pi}(\bar{x})$. This quantity will be calculated via a straightforward subroutine-call to the SUM-7 algorithm. The latter procedure will produce the quantity of 410 (because the elements $\bar{y}_1$ through $\bar{y}_4$ satisfy $e_2^\pi(\bar{x},y)$).

The second step of the SUM-8 algorithm will be designed to produce the quantities of $Z_1^F(\bar{x})$ and $Z_2^F(\bar{x})$. The details of these two calculations are separately discussed in the next two paragraphs.

The SUM-8 algorithm will calculate the value of $Z_1^F(\bar{x})$ by walking the $\mathcal{F}_1(\bar{x})$ set. Note that assumption B stated that this set consisted of the three elements of $\bar{y}_2$, $\bar{y}_3$ and $\bar{y}_6$. For each of these three elements, the SUM-8 algorithm will calculate an associated $\{[e_0^\pi(\bar{x},y)-e_1^\pi(\bar{x},y)]\cdot F(y)\}$ quantity. These three quantities will be equal to respectively zero, 103 and 106 in the present example. The SUM-8 algorithm will add together these quantities during its calculation of $Z_1^F(\bar{x})$. The derived sum of 209 will represent the value of $Z_1^F(\bar{x})$.

A similar procedure will be used during the calculation of $Z_2^F(\bar{x})$. All elements of the $\mathcal{F}_2(\bar{x})$ set will be examined during this calculation. The y-records belonging to this set are $\bar{y}_3\bar{y}_4\bar{y}_5$. These records will have $\{[e_1^\pi(\bar{x},y)-e_2^\pi(\bar{x},y)]\cdot F(y)\}$ values equal to respectively -103, -104 and zero. The SUM-8 algorithm will therefore set $Z_2^F(\bar{x})$ equal to the corresponding sum of -207.

The third step of the SUM-8 algorithm will calculate the value of $S_e^F(\bar{x})$ by using the following formula:

$$S_e^F(\bar{x}) = S^F_{e_2^\pi}(\bar{x}) + Z_1^F(x) + Z_2^F(x) \tag{6}$$

Note that the above sum equals 412.

The SUM-8 algorithm will consequently inform the user that $S_e^F(\bar{x})$ equals this quantity.

One final comment should be made before leaving this example. Some readers may wish to independently check the correctness of the procedure used in this example. This can be done by observing that our present example had $\bar{y}_1\bar{y}_2\bar{y}_3$ and $\bar{y}_6$ constituting the set of y-records that satisfied $e(\bar{x},y)$. The F(y) values of these four records clearly have a sum equal to 412. The procedure in this example thus reached the correct answer.

THEOREM 7.2.M. Let e(x,y) denote an E-8 expression, and let us assume that its Q^F-8(e,R_y) data-image is physically constructed. Under these circumstances, the SUM-8 algorithm will always

I) Correctly calculate the value of $S_e^F(\bar{x})$,

II) Perform this task in $\mathcal{O}(N^e(\bar{x}) + \log^{D(e)}N)$ time.

Proof of Proposition I: The proof of the correctness of the SUM-8 retrieval algorithm will be quite simple. This proof rests mostly on the observation that the SUM-8 algorithm is a trivial extension of the SUM-7 algorithm. The correctness of the SUM-8 algorithm is very easy to verify from this perspective. More specifically, our proof will be based on the following three observations:

1) Theorem 6.4.U, together with the definition of Q^F-8(e,R_y) implies that the first step of the SUM-8 algorithm will correctly calculate the value of $S^F_{e^\pi_K}(\bar{x})$ when it makes a subroutine-call to the SUM-7 algorithm.

2) Definition 7.2.E implies that the $e_i^\pi(\bar{x},y)$ and $e_{i-1}^\pi(\bar{x},y)$ predicates can possess different truth-values only for those y-elements which belong to the $\mathcal{Y}_i(\bar{x})$ set. This fact implies that the second step of the SUM-8 algorithm will correctly calculate its $Z_i^F(\bar{x})$ values (the definition of $Z_i^F(\bar{x})$ can be found in 7.2.I).

3) The definition of $Z_i^F(x)$ together with Remark 7.2.H imply that the third step of the SUM-8 algorithm will correctly calculate the value of $S_e^F(\bar{x})$.

The correctness of the SUM-8 algorithm has thus been verified since all three steps of this procedure have been shown to operate correctly.

Q.E.D.

Proof of Proposition II. *An analysis of the three steps of the* SUM-8 algorithms indicates that

1) The first step of this procedure consumed $\mathcal{O}(\log^{D(e)}N)$ worst-case hashtime.

2) The second step consumed $\mathcal{O}(N^e(\bar{x}))$ time.

3) The third step consumed $\mathcal{O}(K)$ time.

The preceding observations imply that the complete SUM-8 procedure will require $\mathcal{O}(K+N^e(x)+\log^{D(e)}N)$ runtime. Note that K can be regarded as a constant since its magnitude is unrelated to the size of our database. This constant can be omitted in our estimation of runtime order of magnitudes. The SUM-8 algorithm has thus been confirmed to have an $\mathcal{O}(N^e(\bar{x})+\log^{D(e)}N)$ runtime.

Q.E.D.

COROLLARY 7.2.N. The COUNT-8 algorithm will similarly calculate $I_e(\bar{x})$ values in $\mathcal{O}(N^e(x)+\log^{D(e)}N)$ runtime.

Proof: Note that all COUNT algorithms in this thesis can be regarded as those special applications of the general SUM procedures that have F(y) equaling one for all values of y. The preceding

Theorem 7.2.M is thus applicable to the COUNT-8 algorithm. It implies that this algorithm can calculate $I_e(\bar{x})$ in $\mathcal{O}(N^e(\bar{x})+\log^{D(e)}N)$ time.

Q.E.D.

The FIND-8 retrieval algorithm will be our next topic. This procedure will be very similar to the SUM-8 and COUNT-8 retrieval algorithms. The FIND-8 procedure will be formally described in Algorithm 7.2.O. Some readers may prefer to omit that part of this section because it is very similar to the SUM-8 algorithm.

Algorithm 7.2.O. Let $e(x,y)$ denote an E-8 expression, and let us assume that the Q^F-$8(e,R_y)$ data-image is physically constructed. The FIND-8 algorithm will use the following three-step procedure to construct the user's $Y_e(\bar{x})$ sets:

1) The first step of this procedure will consist of a subroutine-call to the FIND-7 algorithm. The called procedure will be instructed to search the Q^F-$8(e,R_y)$ data-image for the purposes of constructing the $Y_{e^{\pi}_K}(\bar{x})$ set.

2) Let $Y(\bar{x})$ denote the set produced by Step 1. The second step of the FIND-8 algorithm will be designed to modify this $Y(\bar{x})$ set so as to make it equal to $Y_e(\bar{x})$. This step will consist of a large DO-LOOP for values of i between 1 and K. The i-th iteration of this DO-LOOP will walk the corresponding $\mathcal{F}_{K+1-i}(\bar{x})$ set. Each y-record belonging to $\mathcal{F}_{K+1-i}(\bar{x})$ will be examined during this walk. Such

a y-record will be inserted into the $Y(\bar{x})$ set when $e^{\pi}_{K-i}(\bar{x},\bar{y})$ = TRUE and $e^{\pi}_{K+1-i}(\bar{x},\bar{y})$ = FALSE. Also, the i-th iteration will delete $\bar{y}$ from $Y(\bar{x})$ when $e^{\pi}_{K-i}(\bar{x},\bar{y})$ = FALSE and $e^{\pi}_{K+1-i}(\bar{x},\bar{y})$ = TRUE.

3) The third step of the FIND-8 algorithm will take advantage of the fact that the user's requested $Y_e(\bar{x})$ set will be produced at the end of the previous DO-LOOP. This step will write that set into the user's workspace.

COROLLARY 7.2.P. The preceding FIND-8 algorithm will correctly produce the user's $Y_e(\bar{x})$ sets in $\mathcal{O}(N^e(\bar{x})+I_e(\bar{x})+\log^{D(e)}N)$ runtime.

Corollary 7.2.P will not be formally proven in this thesis because the FIND-8 and SUM-8 algorithms are very similar. The previous proof of the SUM-8 theorem can be easily modified to justify this corollary.

The last theorem in this section will be Proposition 7.2.Q. That theorem summarizes our previous results plus mentioning nine additional algorithms.

THEOREM 7.2.Q. Let $e(x,y)$ denote an E-8 expression and $e_b(x,y)$ denote the same extended predicate which was mentioned in section 6.7. It will then follow that:

I) The Q^F-8(e,R_y) data-image will enable the SUM-8 and COUNT-8 retrieval algorithms to process predicate $e(\bar{x},y)$ in $\mathcal{O}(N^e(\bar{x})+\log^{D(e)}N)$ worst-case hashtime;

II) The Q^F-8(e,R_y) data-image will enable the FIND-8 algorithm to produce its $Y_e(\bar{x})$ sets in $\mathcal{O}(!_e(\bar{x})+N^e(\bar{x})+\log^{D(e)}N)$ retrieval time;

III) The Q^F-8(e,R_y) data-image will enable the mean-value, existential-quantifier, universal quantifier, and MULTIPLY algorithms to process predicate $e(\bar{x},y)$ in $\mathcal{O}(N^e(\bar{x})+\log^{D(e)}N)$ retrieval-times;

IV) The $Q^F(e_b,R_y)$ data-image will enable the minimum, median, maximum-value, and LOCATE algorithms to process predicate $e(\bar{x},y)$ in $\mathcal{O}(N^e(x)\cdot\log N+\log^{D(e)+2}N)$ retrieval-times;

V) Also data-modification operations in the Q^F-8(e,R_y) and Q^F-8(e_b,R_y) data-images can be performed in respective $\mathcal{O}(\log^{D(e)}N)$ and $\mathcal{O}(\log^{D(e)+1}N)$ runtimes.

Proof: All five propositions in this theorem are simple consequences of the previous theorems. The statements in Propositions I and II require no proof since these results were previously discussed in Theorems 7.2.M, 7.2.P and 7.2.Q.

The proof will therefore begin with Propositions III and IV. These propositions mention seven algorithms for processing E-8

expressions. These seven E-8 algorithms can be executed with procedures that are virtually identical to the procedures that were previously used by the E-7 counterparts of these algorithms. The only distinction between the E-8 and E-7 algorithms is that the former algorithms will make subroutine-calls to SUM-8 and COUNT-8, in those places where the latter algorithms invoked SUM-7 and COUNT-7. It is consequently easy to modify the proofs from sections 6.5 through 6.7 so that they will show that E-8 algorithms satisfy the claims given in Propositions III and IV.

Our attention will now turn to verifying Proposition V. Note that Definition 7.3.G indicated that E-8 and E-7 data-images possess essentially identical data-structures. It thus follows that the same procedures can be used for E-7 and E-8 insertion and deletion operations. The previous E-7 data-modification theorem can thus be used to confirm that the E-8 data-images will support similarly efficient insertion and deletion algorithms.

Q.E.D.

Remark 7.2.R. It should be stated that the runtimes mentioned in Theorem 7.2.Q were extremely conservative worst-case estimates. The reason for this can be seen if Remarks 6.4.V, 6.7.C and 6.7.H are reexamined. Those remarks showed how the E-7 algorithm will often operate in better than the claimed runtimes. The E-8 algorithms are sufficiently similar to the E-7 algorithm for the same better-than-expected runtimes to also apply to the E-8 algorithms.

Remark 7.2.S. Obviously, the E-8 algorithm will benefit from all the coefficient optimization techniques that were mentioned in the earlier parts of this thesis. One additional coefficient optimization technique is exclusively applicable to E-8 expressions. That technique will use the fact that the runtime of the second step of Algorithms 7.2.I and 7.2.O can be improved if those procedures only examined the subsets of $S_i(\bar{y})$ which satisfies the "$e_i(\bar{x},y) \neq e_{i-1}(\bar{x},y)$" inequality. In many applications, straightforward inverted file techniques can be used to help our algorithms efficiently restrict their search space to these subsets.

Remark 7.2.T. It is possible to further generalize this section's theorems for many table-oriented predicates that are more complicated than the preceding tabular predicates. One example of such a predicate would be a $T(x,y,r)$ predicate which assigns a positive truthvalue to an xyr-ordered triple if and only if the associated table contains a triple whose first component equals $\bar{x}$, whose second component equals $\bar{y}$, and whose third component is greater than r. The previous algorithms can be modified so that they can process this table-oriented predicate as well as many other similar predicates. The additional modules required by those predicates are often fairly simple. A full discussion of this topic was omitted for reasons of brevity.

7.3 The Multi-Predicate Redundancy Removal Techniques

In the remainder of this chapter, $e_1(x,y)\ e_2(x,y) \ldots e_K(x,y)$ will denote an arbitrary sequence of K distinct E-8 predicates. This sequence will be studied in detail during the next two sections. Our goal will be to decide how to construct a database that allows the SUM, COUNT, FIND, universal-quantifier, existential-quantifier, mean-value, median-value, maximum-value, and minimum-value algorithms to simultaneously manipulate all K of these predicates in an efficient manner.

One way to perform this multi-predicate task would be to take the straightforward union of the various data-images that are associated with the $e_1 e_2 \ldots e_K$ predicates. Unfortunately, the straightforward union of these data-images will reach an impractical size in many applications. An impractically large database will result whenever there are a large number of distinct $e_1 e_2 \ldots e_K$ predicates. In such cases, the straightforward union is unfeasible because there simply will not be enough memory space to store the data images for all K required predicates.

It clearly will be necessary for a more efficient solution to be found for such multi-predicate applications. A brief survey of some useful multi-predicate optimization techniques will be given in the next two sections of this chapter. That discussion should only be treated as an introduction to multi-predicate queries. A full discussion of this topic will not be included in this thesis because it would be too lengthy. Instead, our goal will be merely to illustrate

the general nature and importance of such techniques.

There will be two topics discussed in the next two sections. This section will explore those special algorithms that remove redundancies from multi-predicate databases. The next section will describe several additional optimizations which can be used in those applications where conservation of memory and data-modification time are more important than economizing data-retrieval time. Our discussion will now begin with a brief description of the notation that will be employed.

The asterisk mark will be used in this section to call attention to those predicates which belong to the E-3 subclass of E-8 expressions. An asterisk will thus be attached to a predicate when that predicate is required to be an E-3 expression. If a predicate has no asterisk then this term will be assumed to be a more general E-8 expression. Our notation will thus have $e_1^* e_2^* \ldots e_j^*$ denote a sequence of E-3 predicates, and it will have $e_1 e_2 \ldots e_K$ denote a sequence of E-8 predicates.

The symbols $u_i(y)$ and $R_{u_i(y)}$ will also be used in our discussion. The former symbol will denote a predicate which is either a y-based unary-term or is the simple predicate of "TRUE." The latter symbol will denote the subset of R_y which satisfies $u_i(y)$.

The preceding terminology will provide a language where all the thesis's data-images can be described. An examination of the previous discussion clearly indicates that all the preceding data-images can be

viewed as finite unions of $Q^F\text{-}3(e_1^*, R_{u_i(y)}; b_{i1} b_{i2} \ldots b_{id})$ data-images. Our new terminology will thus constitute a formalism for describing various optimizations that are possible for these data-images.

During this section, $Q^F\text{-}3(e^*)$ will be called the <u>underlying component</u> of the $Q^F\text{-}3(e^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ data-image. It will be frequently useful to examine these underlying components during the study of data-images. These underlying components will represent the basic data-format which is being iterated throughout the $Q^F\text{-}3(e^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ data-image.

This section will explain how various redundancy-eliminating techniques will enable us to produce an economically sized data-structure that describes the $e_1 e_2 \ldots e_K$ predicates. Such optimizations will take advantage of the fact that the $Q^F(e_1, R_y), Q^F(e_2, R_y) \ldots Q^F(e_K, R_y)$ data-images will usually contain many common data-substructures. Obviously, computer memory will be conserved if the wasted space consumed by these repeating data-structures is eliminated. The redundancy-eliminating algorithms will thus be designed to improve efficiency by removing these repeating data-structures.

There will be many redundancy optimization techniques discussed in the next several paragraphs. These techniques are important because the R_y-description data-structure can very often attain a practical size after the elimination of those redundancies which are shared between the $Q^F(e_1, R_y) Q^F(e_2, R_y) \ldots Q^F(e_K, R_y)$ data-images.

Six general redundancy-eliminating techniques will be briefly

outlined in the next several paragraphs. Some readers may prefer to skim that discussion since the general concept of redundancy elimination is much more important than the details about how such procedures are employed.

Let $e_1(x,y)$ and $e_2(x,y)$ denote two E-8 expressions. The first redundancy elimination technique will take advantage of the fact that the corresponding $Q^F\text{-}8(e_1,R_y)$ and $Q^F\text{-}8(e_2,R_y)$ data-images can each be regarded as a union of several $Q^F\text{-}3(e_i^*,R_{u_i(y)};b_{i1}b_{i2}\ldots b_{id})$ data-images. Often there will be a common data-image belonging to the union sequences of both the $Q^F\text{-}8(e_1,R_y)$ and $Q^F\text{-}8(e_2,R_y)$ structures. Obviously, there is no reason for identical copies of the same data-image to belong to both these sequences. The first redundancy elimination technique will therefore remove such repeating structures.

The preceding redundancy elimination technique will obviously be executed for each of the $e_1e_2\ldots e_K$ predicates that the user wishes to query. The purpose of that process will be to produce a non-repeating sequence of L distinct $Q^F\text{-}3(e_i^*,R_{u_i(y)};b_{i1}b_{i2}\ldots b_{id})$ data-images which satisfies the condition that every one of the user's $Q^F\text{-}8(e_i,R_y)$ structures can be regarded as a union of some subset of these L data-images. <u>A sequence of $Q^F\text{-}3(e_i^*,R_{u_i(y)};b_{i1}b_{i2}\ldots b_{id})$ data-images which satisfies the preceding condition will be called a covering sequence for the $Q^F\text{-}8(e_1,R_y),Q^F\text{-}8(e_2,R_y)\ldots Q^F\text{-}8(e_K,R_y)$ data-images.</u> All the remaining redundancy elimination techniques will presuppose the previous construction of such a covering sequence. They will seek to further optimize this sequence by removing more

subtle redundancies from it.

There are several additional optimizations which can be applied to the covering sequence. One optimizing technique will seek to find members of this sequence which have identical $e^*(x,y)$ and $R_{u(y)}$ components. Suppose $Q^F\text{-}3(e^*,R_{u(y)};b_{i1}b_{i2}\ldots b_{id})$ and $Q^F\text{-}3(e^*,R_{u(y)};b_{j1}b_{j2}\ldots b_{jd})$ are two such members. Let us further assume that the $b_{j_1}b_{j_2}\ldots b_{j_d}$ and $b_{i_1}b_{i_2}\ldots b_{i_d}$ key-sequences are different permutations of the same set of d keys. Such permutated data-images can be regarded to contain equivalent information. A redundancy-removing algorithm should therefore eliminate one of these equivalent data-structures.

There is also a second and more powerful optimization which is possible when two data-images possess the same e^* and $R_{u(y)}$ component. Let $Q^F\text{-}3(e^*,R_{u(y)};b_{i1}b_{i2}\ldots b_{id})$ and $Q^F\text{-}3(e^*,R_{u(y)};b_{j1}b_{j2}\ldots b_{jk})$ denote these two data-images. In many applications, the keys of one of these data-images may be a subset of the other's keys. For example, the second expression's set of b_j keys may be a subset of the first expression's set of b_i keys. In this case, the second expression can be eliminated from the covering sequence because the first expression includes all the information that was contained in the second expression.

Another type of redundancy elimination operation is possible when two members of a covering sequence have the same $R_{u(y)}$ and

$b_1 b_2 \ldots b_d$ components. Let $Q^F\text{-}3(e_i^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ and $Q^F\text{-}3(e_j^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ denote two such structures. In many cases, the predicates e_i^* and e_j^* may satisfy the condition that the $Q^F\text{-}3(e_i^*)$ underlying component contains all the information that is stored in $Q^F\text{-}3(e_j^*)$. In that case, a redundancy eliminating algorithm should eliminate the $Q^F\text{-}3(e_j^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ data-image.

There are other optimizations which are possible when two data-images have the same $R_{u(y)}$ and $b_1 b_2 \ldots b_d$ constituents. Often, the underlying components of these two data-images will contain some common data substructure. In that case, the $Q^F\text{-}3(e_i^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ and $Q^F\text{-}3(e_j^*, R_{u(y)}; b_1 b_2 \ldots b_d)$ data-images should be adjusted so that such repeating data-substructures are eliminated.

It also will often be desirable to adjust the database so that identical sets of pointers do not describe the similar tree structures. Such repeating pointer structures will occur when several terms of the covering sequence have the same $R_{u(y)}$ and $b_1 b_2 \ldots b_d$ components. These similar terms should be processed by a two-step redundancy eliminating algorithm. The first step of this procedure should apply the five optimizations that were outlined in the previous paragraphs. The second step should take those resulting data-images which have common $R_{u(y)}$ and $b_1 b_2 \ldots b_d$ components, and it should merge them into a single $Q^F\text{-}3(e_1^* e_2^* \ldots e_m^*, R_{u(y)}, b_1 b_2 \ldots b_d)$ data-image. The advantage of this merged structure is that it will

eliminate the repeating sequences of pointers that describe similar super-B-trees.

There are many additional possible redundancy-elimination techniques. For example, several optimizations can be produced by combining the previous six operations. Other examples of redundancy-elimination techniques can be produced by generalizing the previous one-predicate coefficient optimization techniques for the cases of multi-predicate queries. These one-predicate techniques were discussed in Remarks 3.6.L, 5.2.N, 5.3.K, 6.2.H, 6.3.N, 6.4.X and 7.2.S. The comments in each of these remarks can be generalized so that they will produce optimizations in a multi-predicate environment.

It should also be stated that all the redundancy-eliminating procedures that were outlined in this section are more complicated than they appear to be. The reason for these added complications can be understood if it is recalled that E-8 data-images are produced by taking the unions of several $Q^F\text{-}3(e^*, R_{u_i(y)}; b_1 b_2 \ldots b_d)$ data-images. The crucial fact about these unions is that there will usually be several different possible sequences of $Q^F\text{-}3(e^*, R_{u_i(y)}; b_1 b_2 \ldots b_d)$ data-structures whose union can be used to form the same E-8 data-image. The difficulty which a redundancy elimination algorithm will face with respect to these unions results from the fact that some decompositions of an initial set of E-8 data-images will lend themselves to much more efficient redundancy elimination operations than others. Special

measures must therefore be taken to assure that a redundancy elimination algorithm uses the most efficient possible decomposition representation of the initial E-8 data-images.

An additional comment should be made in reference to the previous remark. It will obviously be extremely difficult for a database management system to consistently decide which decomposition of the initial E-8 data-images is likely to produce the most efficient redundancy elimination operations. The user should therefore be given the option of overriding the system. If the user exercises this option, then it will be his obligation to specify how the E-8 data-images are to be decomposed into unions of Q^F-3$(e^*, R_{u_1(y)}, b_1 b_2 \ldots b_d)$ structure. In this case, the database management system will perform its redundancy elimination operations on the decomposition sequence which the user has specified.

One final comment should be made about redundancy elimination. The discussion in this section took place in a multi-predicate context where queries were assumed to be made against several different $e_1 e_2 \ldots e_t$ predicates. The proposed techniques are also applicable in certain one-predicate settings. This is because the data-images of E-7 and E-3 expressions are begotten by taking the union of several E-8 data-images (as was shown in Definition 6.4.Q). The proposed redundancy elimination techniques are thus also helpful in removing the repeating data-structures from E-7 and E-8 data-images.

7.4 Multi-Predicate Tradeoff Techniques

A very different type of multi-predicate optimization will be studied in this section. Our attention will be focussed on those applications where the user is willing to accept a "tradeoff" that improves memory space utilization and data-modification time at the expense of data-retrieval time. The ideas in this section are extremely important because they will explain how the general algorithms of this thesis can be usefully applied to many additional types of commercial settings. The next several paragraphs will introduce the terminology that will be used in the forthcoming discussion.

Definition 7.4.A. Let us recall that part 6.4.A of this thesis indicated that there were three kinds of unary predicates that would be discussed in this thesis. (These were the "single-attribute," "double-attribute," and "list-oriented" classes of unary predicates.) Let us also recall that a unary predicate is said to be y-based if it possesses the single variable of y. An E-8 expression will be said to be y-simple if it contains no y-based unary predicates of the list-oriented or double-attribute kind.

Example 7.4.B. Let e(x,y) denote a y-simple E-8 expression. Note that the preceding definition contained no restriction on the x-based unary predicates, xy-equality predicates, xy-order predicates, xy-tabular predicates, or y-based single-attribute unary predicates

which are allowed to appear in $e(x,y)$. Consequently, virtually all E-8 expressions are y-simple. Some of the many examples of y-simple E-8 expressions are shown below. In those examples, $T(x,y)$ denotes a tabular predicate:

(1) $\{x.a_1 = y.b_1 \text{ OR } x.a_2 > y.b_2\}$

(2) $\{x.a_1 = y.b_1 \text{ AND } T(x,y)\}$

(3) $\{x.a_1 > 10 \text{ AND } x.a_1 \geq y.b_1 \text{ AND NOT } x.a_2 = y.b_2\}$

(4) $\{x.a_1 > 10 \text{ AND } y.b_1 > 10 \text{ AND } x.a_1 > x.a_2\}$

Note that Definition 7.4.A indicated that a predicate will fail to be y-simple only if it contains a list-oriented or double-attribute y-based unary predicate. An example of an equation with one of these two types of atomic predicates is shown in equation 5. That equation will fail to be y-simple because of the presence of the "$y.b_1 > y.b_2$" unary predicate:

(5) $\{x.a_1 > y.b_1 \text{ AND } y.b_1 > y.b_2\}$

Definition 7.4.C. Let $e(x,y)$ denote an E-8 expression. This expression will be said to have a y-attribute set equal to $b_1 b_2 \ldots b_d$ if that sequence represents the full set of y-attributes that appear in $e(x,y)$.

Example 7.4.D. Clearly, equation 6 has a y-attribute set equal to $b_1 b_2$:

(6) $\{x.a_1 = y.b_1 \text{ AND } x.a_2 > y.b_2 \text{ AND NOT } x.a_3 = y.b_2\}$

Definition 7.4.E. Let $e(x,y)$ denote a y-simple E-8 expression. The symbol of $D^{\#}(e)$ will denote the size of this expression's y-attribute set. In other words, $D^{\#}(e)$ will denote the number of different y-attributes appearing in $e(x,y)$.

Example 7.4.F. The previous equation 6 had a $D^{\#}(e)$-value equal to two because it contained the two y-attributes of b_1 and b_2.

Observation 7.4.G. It is instructive to compare the definition of $D^{\#}(e)$ and $D(e)$. Note that the former quantity counted all the y-attributes in e whereas the latter term only counted the y-attributes in e's order predicates. The larger number of attributes that are counted by $D^{\#}(e)$ clearly implies that $D^{\#}(e) \geq D(e)$. The preceding inequality will play an important role in the rest of this section. One of the main themes of this section will be that the user can attain significant savings in memory space and data-modification time if he is willing to accept a tradeoff where the data-retrieval time is increased by replacing its $\log^{D(e)} N$ component with a slightly larger $\log^{D^{\#}(e)} N$ quantity.

One more definition must be introduced before a more detailed description can be given to our tradeoff techniques. The next paragraph will define the $Q^F(R_y; b_1b_2\ldots b_d)$ data-image. This data-image will constitute the main memory structure that will be studied in this section:

Definition 7.4.H. The symbol of $Q^F(R_y; b_1b_2\ldots b_d)$ will be an abbreviation for Q^F-8("TRUE," R_y, $b_1b_2\ldots b_d$). In other words, $Q^F(R_y; b_1b_2\ldots b_d)$ will denote that special data-image which is built around the predicate of "TRUE," the R_y relation, and the $b_1b_2\ldots b_d$ keys.

Much of this section will be devoted to the study of the $Q^F(R_y; b_1b_2\ldots b_d)$ data-image. Our goal will be to develop an algorithm that enables this data-image to answer any query about any y-simple E-8 predicates whose y-attribute set is either equal to $b_1b_2\ldots b_d$ or is equal to a subset of $b_1b_2\ldots b_d$. Throughout this section, it will be emphasized that there are both advantages and disadvantages to performing such searches with the $Q^F(R_y; b_1b_2\ldots b_d)$ data-image. The advantage of this data-image will be that it will often help conserve memory-space and data-modification time. The disadvantage to performing retrievals with the $Q^F(R_y; b_1b_2\ldots b_d)$ data-image will be that such searches will usually have slower retrieval-times than those produced by earlier algorithms from other parts of this thesis.

Several comparisons will be made in this section between the Q^F-8(e, R_y) and $Q^F(R_y; b_1b_2\ldots b_d)$ data-images. Note that the earlier sections of this thesis showed how the former data-image can be used to support any one of the FIND, SUM, COUNT, universal-quantifier, existential-quantifier, mean-value, median-value, minimum-value, maximum-value or LOCATE queries for E-8 expressions. The previous search algorithms that used this Q^F-8(e, R_y) data-image will be henceforth called the algorithms of the "Predicate-Oriented" search method. The main purpose of this section will be to develop a second alternative search-method that will instead use the $Q^F(R_y; b_1b_2\ldots b_d)$ data-image for performing the *same ten types of queries*. This *second search-method* will be henceforth called the "Reduction-Oriented" method.

A formal description of the procedure used by the Reduction-Oriented search method will be given in Algorithm 7.4.I. This paragraph will contain a more intuitive explanation of the procedure that is used by this search method. In most respects, the Reduction-Oriented search method will be identical to the Predicate-Oriented search method. The Reduction-Oriented method will differ from the Predicate-Oriented method only in its treatment of xy-equality and y-based single-attribute unary predicates. The Reduction-Oriented algorithms will process these two types of atomic predicates by converting them into equivalent sequences of two-variable order predicates. The purpose of that transformation will be to convert an initial e(x,y) predicate into an equivalent form that allows searches in the

$Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image to be performed via subroutine-calls to the Predicate-Oriented algorithms. Theorem 7.4.J will demonstrate the success this approach has in processing y-simple E-8 expressions. That theorem will state that the only difference between the retrieval-times of the Reduction-Oriented and Predicate-Oriented search methods is that the retrieval times of the former algorithms will contain a $\log^{D'(e)} N$ quantity in those places where the retrieval-time of the latter algorithms contained a $\log^{D(e)} N$ quantity.

One final introductory comment should be made before the Reduction-Oriented search method is discussed in more detail. Note that Remark 7.4.G indicated that $D'(e) \geq D(e)$. This inequality will obviously imply that the $\log^{D'(e)} N$ retrieval-time of the Reduction-Oriented algorithms will always be at least as expensive as the $\log^{D(e)} N$ time of the Predicate-Oriented algorithms. Throughout the forthcoming discussion, the reader should bear in mind that <u>a worse retrieval-time is accepted for the Reduction-Oriented algorithms because a resulting tradeoff will produce economies in memory-space and data-modification time</u>. The formal discussion of the Reduction-Oriented search method will begin with the following algorithm description:

<u>Algorithm 7.4.I</u>. Let us assume that the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image is physically constructed. Let $e(x,y)$ denote some y-simple E-8 predicate whose y-attribute set is either equal to $b_1 b_2 \ldots b_d$ or is a subset of $b_1 b_2 \ldots b_d$. The Reduction-Oriented

algorithm will use the following two-step procedure when it is searching the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image with the $e(x,y)$ predicate for the purposes of performing a FIND, SUM, COUNT, universal-quantifier, existential-quantifier, mean-value, median-value, minimum-value, maximum-value, or LOCATE query:

1) The first step of the Reduction-Oriented algorithm will be designed to convert e's initial xy-equality and y-based single-attribute unary predicates into equivalent sequences of two-variable order predicates. During the discussion of this conversion process, c will denote a constant, and a and b will denote respective attributes of x and y. The following three rules will be used for performing these conversions:

   1a) Each xy-equality predicate similar to "x.a = y.b" will be converted into an equivalent order predicate phrase of the form: "x.a ≥ y.b AND x.a ≤ y.b."

   2a) Each unary predicate similar to "y.b > c" will be replaced by an equivalent order predicate that compares y.b to a variable that has the value of c stored in it.

   1c) Each unary term similar to "y.b = c" will be processed by a procedure that combines the previous two techniques. The first step of this procedure will convert the initial "y.b = c" term into an

equivalent "y.b ≥ c AND y.b ≤ c" phrase. The second step will treat c as if it was a variable during the processing of this phrase.

2) The second step of the Reduction-Oriented algorithms will take the predicate produced by the previous conversion step, and it will make a subroutine-call to one of the Predicate-Oriented algorithms for the purposes of completing the processing of e(x,y). The specific Predicate-Oriented algorithm that is called by this step will depend on whether the user has made a request for a FIND, SUM, COUNT, universal-quantifier, existential-quantifier, mean-value, median-value, minimum-value, maximum-value or LOCATE query. This step will execute the user's command by making a subroutine-call to the appropriate Predicate-Oriented algorithm and by instructing that procedure to execute the user's request by searching the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image with the predicate that was produced by step 1.

THEOREM 7.4.J. Let e(x,y) denote a y-simple E-8 expression whose y-attribute set is either equal to $b_1 b_2 \ldots b_d$ or is a subset of $b_1 b_2 \ldots b_d$. Let us further assume that the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image is physically constructed. Under these circumstances, the ten Reduction-Oriented search algorithms will perform their retrieval operations in runtimes which are identical to the Predicate-Oriented

algorithms except that the $\log^{D(e)} N$ terms are replaced by the $\log^{D^{\#}(e)} N$ terms. A more specific description of the retrieval-times of our Reduction-Oriented algorithms can be given if $N^e(\bar{x})$ denotes the same quantity which it designated in section 7.3. (In other words, $N^e(\bar{x})$ will equal the number of y-elements that are in the xy-tables which are referred to by predicate e and element $\bar{x}$.) Under these circumstances, the following three rules will summarize the runtimes that are attained by the various Reduction-Oriented algorithms:

I) The SUM, COUNT, universal-quantifier, existential-quantifier, and mean-value algorithms will process predicate e(x,y) in $\mathcal{O}(N^e(\bar{x}) + \log^{D^{\#}(e)} N)$ retrieval-time.

II) The FIND algorithm will produce the $Y_e(\bar{x})$ sets for predicate $e(\bar{x},y)$ in $\mathcal{O}(I_e(\bar{x}) + N^e(\bar{x}) + \log^{D^{\#}(e)} N)$ retrieval-time.

III) The maximum-value, median-value, minimum-value, and LOCATE algorithms can perform a query for one of the $b_1 b_2 \ldots b_d$ attributes and the e(x,y) predicate in $\mathcal{O}(N^e(\bar{x}) \cdot \log N + \log^{D^{\#}(e)+2} N)$ retrieval-time.

The proof of Theorem 7.4.J will deliberately be kept very short because the Reduction-Oriented algorithms are mathematically simple extensions of the previous Predicate-Oriented algorithms. Also, for the sake of brevity, the proof will verify only the first of the three propositions given in Theorem 7.4.J. Propositions II and III will not

be formally proven in this section because their proofs would be very similar to the following proof of proposition I:

Proof of Proposition I: Note that the three transformation rules in Step 1 of the Reduction-Oriented algorithms will convert an initial e(x,y) predicate into a logically equivalent expression that is built out of xy-order, xy-tabular, and x-based unary predicates. Let $b_{j_1} b_{j_2} \ldots b_{j_K}$ denote the y-attribute set of e(x,y). It is fairly easy to use the techniques from Chapter 6 and section 7.2 to confirm that the Predicate-Oriented algorithms can perform SUM, COUNT, universal-quantifier, existential-quantifier, and mean-value queries in $\mathcal{O}(N^e(\bar{x}) + \log^{D^\theta(e)} N)$ retrieval-time when they are searching the $Q^F(R_y; b_{j_1} b_{j_2} \ldots b_{j_K})$ data-image with the predicate that is produced by the just-mentioned conversion rules.

The implications of the preceding fact can be understood if the hypothesis of Theorem 7.4.J is examined. That hypothesis stated that e's y-attribute set of $b_{j_1} b_{j_2} \ldots b_{j_K}$ is a subset of $b_1 b_2 \ldots b_d$. This fact can be easily shown to imply that the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image will support all the search functions which are supported by the weaker $Q^F(R_y; b_{j_1} b_{j_2} \ldots b_{j_K})$ data-image. The $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image must thus support the same SUM, COUNT, universal-quantifier, existential-quantifier and mean-value search queries which the last paragraph had shown that the $Q^F(R_y; b_{j_1} b_{j_2} \ldots b_{j_d})$ data-image would support. Proposition I has thus been proven since we have shown that

the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image allows the user to perform SUM, COUNT, universal-quantifier, existential-quantifier, and mean-value queries for predicate e(x,y) in $\mathcal{O}(N^e(\bar{x}) + \log^{D^\theta(e)} N)$ retrieval-time. Similar reasoning can be used to prove Propositions II and III.

Q.E.D.

Remark 7.4.K. Note that the Reduction-Oriented method was designed to perform searches only in $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-images. Let $R_{u_i(y)}$ once again denote the subset of R_y that satisfies the $u_i(y)$ condition. It is fairly easy to modify the Reduction-Oriented method so that it can also search data-structures that consist of the unions of several $Q^F(R_{u_i(y)}; b_1 b_2 \ldots b_d)$ data-images. The advantage of such an expanded version of the Reduction-Oriented method is that it will enable this method to also process those E-8 expressions which are not y-simple. A detailed description of such extensions of the Reduction-Oriented method was omitted in this thesis in order to keep our discussion brief.

Remark 7.4.L. It should also be stated that the retrieval-times given in Theorem 7.4.J were extremely conservative worst-case estimates. In many applications, the Reduction-Oriented algorithms will operate in runtimes that are far better than these worst-case estimates. Examples of such applications can be constructed with the techniques that were previously outlined in Remarks 6.4.V, 6.7.C and 6.7.H.

Remark 7.4.M. The runtime coefficient of the Reduction-Oriented retrieval algorithms should be briefly mentioned. Let $b_{j_1} b_{j_2} \ldots b_{j_K}$ denote the y-attribute set of $e(x,y)$, and let L_j denote the number of order predicates in $e(x,y)$ that use the $y.b_j$ attribute (after transformation rules 1a through 1c are applied). Under these circumstances, the coefficients in front of the $\log^{D^{\#}(e)} N$ terms of our ten Reduction-Oriented retrieval algorithms will be proportional to $L_{j_1} \cdot L_{j_2} \cdot \ldots \cdot L_{j_K}$. These coefficients can be easily seen to have a very small and practical size for the simple predicates that a commercial user is likely to make.

Remark 7.4.N. Note that the Reduction-Oriented algorithms used the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image for storing its main data-structures. There are two comments which should be made about this data-image. These are that:

1) Theorem 6.4.U implies that the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image will support $\mathcal{O}(\log^d N)$ insertion and deletion data-modification operations.

2) Remarks 6.3.N and 6.3.O discuss certain optimizations that will enable the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image to consume $\mathcal{O}(N \log^{d-1} N)$ memory space.

It should also be stated that the coefficient behind the preceding data-modification and memory-space estimates are fairly small for most values of d.

Remark 7.4.O. Note that $D^{\#}(e)$ will be less than or equal to d when predicate $e(x,y)$ has a y-attribute set that is a subset of $b_1 b_2 \ldots b_d$. This observation together with the preceding remark and Theorem 7.4.J imply that the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image will occupy $N \log^{d-1} N$ storage space and that it will support data-retrieval and modification operations that can be performed in $\log^d N$ worst-case time. These estimates of expended computer resources are important because they indicate that the future hardware technologies of the 1980's will make the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image practical to use in most applications where d is less than or equal to 4.

Remark 7.4.P. It must be emphasized that the main difference between the Reduction-Oriented search method and the Predicate-Oriented method is the efficiency with which each method performs its task. These two methods were designed to efficiently solve two different types of problems. This remark will contain four comparisons between the Reduction-Oriented and Predicate-Oriented search method. The purpose of that discussion will be to illustrate the different types of circumstances where each search method is preferable to employ:

1) The Reduction-Oriented method and its associated $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image should be used in most multi-predicate applications where the user wishes to conserve memory and data-modification time for a collection of predicates whose y-attribute sets are a subset of $b_1 b_2 \ldots b_d$. Let $e_1 e_2 \ldots e_K$ denote this set of predicates.

The advantage of the Reduction-Oriented method in this situation is that it will allow the user to replace the K distinct $Q^F(e_i, R_y)$ data-images with one single $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image. Such a substitution will usually produce savings in memory space and data-modification time.

2) A second advantage of Reduction-Oriented method is that it will allow the user to process predicates which he may not have anticipated at the time when his database was originally designed. The $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image of the Reduction-Oriented method can thus be applied to <u>any</u> y-simple E-8 predicates whose y-attribute set is a subset of $b_1 b_2 \ldots b_d$. This fact is important because it means that the user's database will require no modification if an unanticipated predicate query of this type arises. Many users will greatly appreciate the Reduction-Oriented method because of its ability to so easily process such unanticipated predicates.

3) <u>The Reduction-Oriented method should usually be avoided when the user is processing only one single isolated predicate</u>. In this case, the Predicate-Oriented method will usually be more efficient than the Reduction-Oriented method from each of the three criteria of data-retrieval time, data-modification time, and memory space. This is because <u>the</u>

<u>Q^F-$\mathcal{B}(e, R_y)$ data-image will usually offer a more efficient representation for describing one single isolated predicate than is possible under the $Q^F(R_y; b_1 b_2 \ldots b_d)$ data-image</u>.

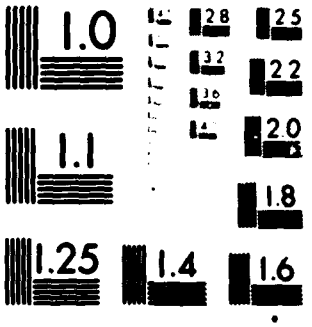
4) <u>The Predicate-Oriented method should always be used when the user wishes to minimize data-retrieval time</u>. The Reduction-Oriented method is not applicable under such circumstances because it would cause the retrieval-time's $\log^{D(e)} N$ component to be replaced by a more expensive $\log^{D^{\#}(e)} N$ quantity.

<u>Remark 7.4.Q</u>. In certain applications, it will be desirable to use a search method that employs a strategy which is intermediate between the Predicate-Oriented and Reduction-Oriented strategies. Such a "hybrid" search method will apply the Reduction-Oriented conversion rules to some (but not all) of e's xy-equality and y-based one-attribute unary predicate. This type of hybrid algorithm should be designed to carefully analyze each one of e's atomic predicates and to apply the Reduction-Oriented conversion rules in only those instances where a definite gain in efficiency results. Hybrid algorithms are important because they will often attain a better combined utilization of memory space, data-modification time, and data-retrieval time than is possible under either the Reduction-Oriented or Predicate-Oriented methods. A detailed description of how hybrid algorithms should decide when to apply a Reduction-Oriented conversion rule

AD-A110 139 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 5/2
PREDICATE-ORIENTED DATABASE SEARCH ALGORITHMS.(U)
MAY 78 D E WILLARD N00014-76-C-0914
UNCLASSIFIED TR-20-78 NL

3 of 3
AD
A110139

END
DATE
FILMED
8-82
DTIC



MICROCOPY RESOLUTION TEST CHART

will not be given in this thesis for the sake of brevity. Instead, this remark will only include a summarized description of three useful principles that should be frequently followed by a hybrid algorithm. These three principles are mentioned below:

1) It will often be useful for a hybrid algorithm to distinguish those y attributes which appear in e's xy-order predicates from those which do not. This is because it will be frequently beneficial for a hybrid algorithm to apply the Reduction-Oriented conversion rules only to those xy-equality and y-based single-attribute unary predicates whose y-attributes appear in one of e's xy-order predicates.

2) It also often will be beneficial for a hybrid algorithm to identify common phrases which are shared by several predicates which the database management system is processing. This is because performance will often be improved if the hybrid algorithm refrains from applying the Reduction-Oriented conversion rules to the atomic predicates which are contained in these common phrases.

3) A hybrid algorithm should also consider section 7.3's redundancy-removing techniques when it is deciding whether or not to apply a Reduction-Oriented conversion rule. Often, a judicious application of these conversion rules will cause the redundancy removing techniques to produce many additional optimizations.

There is also one other multi-predicate optimization technique which should be briefly mentioned before the close of this section. This technique will be called the "Containment Search" method. The following definition will be useful in the discussion of this search method.

<u>Definition 7.4.R</u>. Let $e^*(x,y)$ and $e(x,y)$ denote two predicates. The predicate $e^*(x,y)$ will be said to contain $e(x,y)$ if it can be logically proven that the set of xy-ordered pairs that satisfy $e^*(x,y)$ must necessarily contain the set of ordered pairs which satisfy $e(x,y)$.

The preceding concept of containment will be useful in certain applications where the user wishes to conserve memory-space and data-modification time. An example of such an application can be given if it is assumed that the predicate $e^*(x,y)$ contains $e(x,y)$. Let us further assume that the $\psi^{\mathbf{F}}(e^*, R_y)$ data-image is physically constructed and that the user wishes to avoid constructing additional data-structures for processing predicate $e(x,y)$. In most such cases, it will be necessary for the user to employ a "containment method" for query predicate $e(\bar{x},y)$. Such a method will use a two-part procedure for answering queries about $e(\bar{x},y)$. The first part of this method will use one of the previous FIND algorithms to construct the set of y-records that satisfy the "containing" $e^*(\bar{x},y)$ condition. The second part of this procedure will answer the user's specific query about the "contained" $e(\bar{x},y)$ predicate by making an exhaustive search through the previous set.

Remark 7.4.S. There are two versions of the containment method that will be useful to employ. The difference between these two versions is that the first will rely upon the Predicate-Oriented method and the second upon the Reduction-Oriented method during their invocations of the FIND algorithm. These two versions will have respective retrieval-times of $\mathcal{O}(I_{c^*}(\bar{x}) + N^{c^*}(\bar{x}) + \log^{D(c^*)}N)$ and $\mathcal{O}(I_{c^*}(\bar{x}) + N^{c^*}(\bar{x}) + \log^{D(c)}N)$. The $I_{c^*}(\bar{x})$ component in these retrieval-times will cause the containment method to usually have less efficient retrieval-times than that of the previous search algorithms. The containment method thus should only be applied in those applications where either $I_c(\bar{x})$ has a small value or where the requirement of conserving memory-space and data-modification time precludes the usage of any other method.

Remark 7.4.T. In some applications, the containment method will be able to use any one of several c_i predicates that contain $c(x,y)$. In these cases, runtime will be minimized if the containment method employs that $c_i^*(\bar{x},y)$ predicate which has the lowest $I_{c_i^*}(\bar{x})$ value. This predicate can be located with the aid of the COUNT algorithm.

Remark 7.4.U. Let $c^*(x,y)$ once again denote a predicate which contains $c(x,y)$. In certain applications, the user will be able to avoid the containment method when he is searching the $Q^F(c^*,R_y)$ data-image with the $c(x,y)$ predicate. This is because special conditions will sometimes arise that will allow the Reduction-Oriented or

Predicate-Oriented algorithms to search the $Q^F(c^*,R_y)$ data-image with the $c(x,y)$ predicate.

Remark 7.4.V. It must be emphasized that the purpose of the last two sections of this chapter was to outline the many options that are available to a user when he employs the algorithms of this thesis. Obviously, it will be necessary for a new computer language to be designed so that the user can have access to these many combinatorics techniques. Such a computer language should have primitives that allow the user to either specify his own data-image and search method or to rely upon a database optimizer to make these decisions. The main purpose of this thesis has been to outline the underlying combinatorics techniques which the primitives of this language should be designed to access.

7.3 Further Extensions

The goal of this section will be to introduce further terminology and to use this notation to describe two major extensions that can be added to the algorithms that were proposed in this thesis. There will be no attempts at formal proofs or detailed descriptions of algorithms in this section. Such a discussion will be omitted because it would require at least one hundred additional pages. Instead, the objective of this section will be merely to announce several important theorems that I have proven. The papers that contain the lengthy proofs of these theorems will be written and published at a later time.

The first topic in this section will be the optimization techniques that are possible when a large number of queries are made against a single predicate. The nature of this type of optimization can best be illustrated if the example is considered where R_y has $\Theta(N)$ distinct y-elements and where the $S_e^y(\bar{x}_i)$ values for the predicate e and for the N arguments of $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ need to be immediately calculated. Such calculations of the N different $S_e^y(\bar{x}_i)$ values could, of course, be performed by N corresponding subroutine-calls to the SUM-S algorithm. Each individual invocation of the SUM-S algorithm would perform its operations in the previously proven $\Theta(N^e(\bar{x}_i)+\log^{D(e)}N)$ retrieval-time. Let N^e denote the sum of the $N^e(\bar{x}_i)$ values of the N distinct elements of $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$. The combined time required by all N invocations of the SUM-S algorithm would then possess an $\Theta(N^e+N\log^{D(e)}N)$ magnitude. This runtime will attain the slightly

smaller magnitude of $\Theta(N\log^{D(e)}N)$ in those applications where $e(x,y)$ is an E-1 expression.

It is obvious that the $N\log^{D(e)}N$ components in the preceding runtimes will attain a prohibitively expensive order of magnitude in many applications. The first three theorems in this section will state that it is often possible to attain better runtimes. These theorems will, in other words, describe certain optimizations which are applicable when the user is performing simultaneous queries for a large set of arguments such as $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$. These optimizations will enable the $N\log^{D(e)}N$ components in the previous runtimes to be replaced by smaller quantities. The next several definitions will introduce the terminology that will enable us to describe these more efficient runtimes.

Definition 7.3.A. Let a_i denote some attribute of x. The symbol of $D^e(e,a_i)$ will denote the number of distinct y-attributes which appear in the subset of e's xy-order predicates that do not possess an x-attribute of a_i.

Definition 7.3.B. Let $e(x,y)$ denote an E-S expression and let $a_1a_2\ldots a_K$ denote the set of x-attributes that appear in this expression. The symbol $D^e(e)$ will denote the minimum value that is attained by the associated $D^e(e,a_i)$ values.

Example 7.3.C. Let $e(x,y)$ denote the following E-S expression:

(1) $\{x.a_1 > y.b_1 \text{ OR } x.a_2 > y.b_2 \text{ OR } x.a_3 > y.b_3 \text{ OR } x.a_1 > y.b_2 \text{ OR } x.a_1 < y.b_3\}$

In this example, $D^*(e,a_1)$ will equal 2 because the order predicates that do not involve $x.a_1$ possess the 2 attributes of b_2 and b_3. Also, the values of $D^*(e,a_2)$ and $D^*(e,a_3)$ will equal 3 in this example. Note that the smallest of the three $D^*(e,a_i)$ quantities has a value of 2, $D^*(e)$ will thus equal this number.

The concept of $D^*(e)$ is important because the $\log^{D(e)}N$ runtime component in the Predicate-Oriented retrieval algorithms can be replaced by a more efficient $\log^{D^*(e)}N$ component if certain special $N \log N$ sorting operations are invoked. Such sorting operations will improve the overall efficiency of our retrieval algorithms in those applications where the user desires to immediately perform queries for a large number of $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ arguments and where additionally $D(e) > 2$. The proof that such efficiencies are possible under these circumstances will be given in a subsequent paper. In that paper, it will be assumed that R_y is a relation whose size has the same order of magnitude as N and that N^* denotes the sum of the $N^*(\bar{x}_i)$ values for the $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ elements. Under these assumptions, the following three theorems will be proven:

THEOREM 7.5.D. The SUM, COUNT, MULTIPLY, universal-quantifier, existential-quantifier and mean-value algorithms can perform their calculations for all N of the user's $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ elements in one single task that consumes $\mathcal{O}(N^* + \log^{D(e)}N + N \log N)$ worst-case hashtime.

THEOREM 7.5.E. The minimum-value, median-value, maximum-value, and LOCATE algorithms can similarly perform their procedures for elements $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ in $\mathcal{O}(N^* \log N + \log^{D^*(e)+2}N + N \log N)$ worst-case hashtime.

THEOREM 7.5.F. If I_e denotes the sum of the $I_e(\bar{x})$ values for elements of $\bar{x}_1\bar{x}_2\ldots\bar{x}_N$ then the FIND algorithm can produce the $Y_e(\bar{x}_i)$ for all N of these elements in $\mathcal{O}(N^* + I_e + \log^{D^*(e)}N + N \log N)$ worst-case hashtime.

<u>Remark 7.5.G</u>. It should be stated that the runtimes which were mentioned in the preceding theorems were extremely conservative worst-case estimates. My future paper will thus show that each of the preceding algorithms can frequently operate in much better than the claimed runtimes for the same reason that the other algorithms in this thesis also often exceeded their claimed runtimes. It should also be stated that the only reason that the discussion of Theorems 7.5.D, 7.5.E and 7.5.F has been deferred to a later paper is that it simply would have taken too much time for me to write up the formal proofs.

There is also a second major topic that has been mostly omitted from this thesis for the sake of brevity. That topic is the applications of the preceding algorithms to Codd's relational calculus language.

The general nature of Codd's relational calculus language was briefly described in section 2.2 of this thesis. That section indicated that the relational calculus had a grammar similar to the predicate

calculus and that the purpose of this language was to enable the user to make queries on finite sets with the existential and universal quantifiers.

A more detailed description of relational calculus language can be given if the meaning of the free variables in this language are explained. In Codd's language, a variable to left free of a quantifier when the user is trying to retrieve the elements which that variable is pointing to. Throughout this section, it will be assumed that the user designates a free variable by inserting the symbol of "F" and the name of the free variable to the left of the universal and existential quantifiers in the relevant relational calculus expression. The expression of "$F_x F_y \forall_z \exists_w \alpha(x,y,z,w)$" will thus indicate that the variables of x and y are free and that the user wishes to retrieve those xy-ordered pairs in the cross-product of R_x and R_y which satisfy the "$\forall_z \exists_w \alpha(x,y,z,w)$" condition.

One final introductory comment should be made about the free variable designators. The theorems in this section will treat these designators in the same manner that they treat universal and existential quantifiers. Consequently, the term "quantifier" in this section will refer to a universal quantifier, existential quantifier, or free variable designator.

There will be no restriction in the number of variables that will be allowed to appear in the relational calculus expressions that will be discussed in this section. The symbols of $V_1 V_2 \ldots V_k$ will therefore

be used to denote the variables that appear in these expressions. These variables will be said to form "relational graphs." This concept is defined below:

<u>Definition 7.5.H</u>. Let r denote a relational calculus expression that uses the variables of $V_1 V_2 \ldots V_k$. The symbol of "G(r)" will denote the "relational graph" that is associated with this expression. The vertices of this graph will be defined to be the variables of $V_1 V_2 \ldots V_k$. The G(r) graph will be defined to have a directed edge from vertex V_i to vertex V_j if and only if the following two conditions hold:

1) The quantifier for variable V_i must appear to the left of V_j's quantifier in expression r;
2) There must also be some atomic predicate in expression r which uses the variables of V_i and V_j.

The main theorem in my future paper on the relational calculus will state that any relational calculus expression which has a tree or forest graph structure can be efficiently processed through a series of subroutine-calls to this thesis's E-S algorithms. A more precise description of this procedure can be given after the following definitions are introduced.

<u>Definition 7.5.I</u>. Let r denote a relational calculus expression, and let x and y denote two variables such that x appears to the left of

y in expression r. Let us further assume that a_i is an attribute of x. The symbol $D^*(r,x,y,a_i)$ will henceforth denote the number of distinct y-attributes that appear in the subset of r's xy-order predicates which do not contain the x-attribute of a_i.

Definition 7.5.J. Let r, x, and y have the same meanings as in Definition 7.5.I. Let $a_1a_2\ldots a_K$ denote the set of attributes of variable x. The symbol of $D^*(r,x,y)$ will denote the minimum $D^*(r,x,y,a_i)$ value that is attained by these attributes.

Comment: The above definitions of $D^*(r,x,y,a_i)$ and $D^*(r,x,y)$ are obviously very similar to the previous definitions of $D^*(o,a_i)$ and $D^*(o)$ that were given in 7.5.A and 7.5.B. This is because the proposed relational calculus algorithm will make subroutine-calls to the algorithms that were described in Theorems 7.5.D, 7.5.E, and 7.5.F. The following additional definition will help further define the final runtime that is produced by such a process:

Definition 7.5.K. Let r denote a relational calculus that uses the variables of $V_1V_2\ldots V_K$. Let V_i denote a variable that appears to the left of V_j in expression r, and let $D^*(r,V_i,V_j)$ denote that quantity suggested by Definition 7.5.J. The symbol of $D(r)$ will denote the maximum value which is attained by all the $D^*(r,V_i,V_j)$ values that are generated by the $V_1V_2\ldots V_K$ variables.

The main theorems in my forthcoming paper will state that any relational calculus expression with a tree or graph structure can be

processed in approximate $N\log^{D(r)}N$ runtime. A more precise description of the proposed runtimes can be given if the following notation is introduced:

1) The symbol N will be used in the following discussion to denote the magnitude of all the relations and tabular-predicate-tables referred to in expression r.

2) The symbol of t_r will denote the number of types that satisfy relational calculus expression r.

The following two theorems will use the preceding notation to describe the runtimes which can be attained by relational calculus algorithms:

THEOREM 7.5.L. Let r denote a relational calculus expression which is built out of the usual equality, order, unary, and tabular atomic predicates and which additionally has a tree or forest graph structure. The t_r tuples that satisfy this expression can be found and fully listed in $\mathcal{O}(t_r + N\log^{D(r)}N + N\log N)$ retrieval-time.

THEOREM 7.5.M. Let us assume that the expression r in the previous theorem possesses no two-variable order predicate. In that case, the set of t_r distinct tuples satisfying r can be produced in the somewhat more efficient $\mathcal{O}(t_r + N)$ retrieval-time.

Remark 7.5.N. The preceding theorems will be proven in my future paper on relational calculus expressions. That paper will also explain how the preceding theorems can be applied to many relational calculus expressions that do not represent a tree or forest-like relational graph. This is because many expressions of this kind can be converted into equivalent expressions that possess a tree or graph structure. Such conversions techniques are important because they will enable most commercial users to execute their relational calculus expressions in N log N runtime.

The paper that will contain the proofs of the claims that were made in this section will be written sometime in the coming year. The reason these proofs were omitted from this thesis is that they simply would have taken me too much time to formally write up.

BIBLIOGRAPHY

AC-75 Astrahan, M. M. and Chamberlin, D. D., "Implementation of a structured English query language," Comm. ACM 18, 10 (Oct. 1975), pp. 580-588.

AHU-74 Aho, A. V., Hopcroft, J. E. and Ullman, J. F., "The Design and Analysis of Computer Algorithm," Addison-Wesley Publishing Co., Reading, Mass., 1974.

AVL-61 Adel'son-Velskiy, G. M. and Landis, Y. M. An algorithm for the organization of information. Doklady Akad. Nauk, SSSR (Moscow) 146, 2 (1962), 263-266. Translation, CFSTI, JPRS, 17, Dept. of Commerce, Washington, D. C.

BCE-77 Bleugen, M., Ceary, R., and Ramarao, K. An encoding method for multi-field sorting and indexing. Comm ACM, V 20 (11 (Nov), 1977.

BE-77 Bleugen, M. W. and Ramarao, K. P. Storage and access in relational data bases. IBM Systems Journal V 16 4th Quarter, 1977, pp. 363-378.

Berkeley-75 Held, G. D., Stonebraker, M. R., and Wong, E. "Ingres: a relational data base system," Proc. AFIPS National Computer Conf., May 1975, Vol. 44, AFIPS Press, Montvale, N. J., 1975, pp. 409-416.

Boehm-72 Report P-4795 Rand Corp., Santa Monica, California, 1972.

Chamberlin-76 "Relational data-base management system," Computing Surveys 8, No. 1, 43-66 (March 1976).

Chiba-75 "A data base search algorithm based on complicated retrieval conditions," The Sohm Kiyo 5, (1975) pp. 139-175.

Codd-70 "A relational model for data for large shared data banks," Communications of the ACM 13, No. 6 377-387 (June 1970).

Codd-71a "A data base sublanguage founded on the relational calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, Nov. 1971," ACM, New York, 1971, pp. 35-68.

Codd-71b "Relational algebra," Courant Computer Science Symposia 6; "Data Base Systems," New York, May 1971, Prentice-Hall, New York, 1971.

Codd-75 "Implementation of relational data base management systems," FDT, Quarterly Bulletin of ACM-SIGMOD 7, 3-4 (1975).

Date-77 "An Introduction to Database systems," Addison-Wesley, 1977. (This edition of Date's book is much more comprehensive than 1975 edition.)

England-76
Todd, S. J. P., "The Peterlee Relational Test Vehicle - a system overview," IBM Systems Journal 15, No. 4, 285-308 (1976).

Gotlieb-75
"Computing joins of relations," Proc. ACM-SIGMOD Conf. May 1975," ACM, New York, 1975.

HSW-75 This article is the same as Berkeley-75.

Knuth-73 The Art of Computer Programming Volume 3, Addison-Wesley, Reading, Mass.

LHM-70 Multi-attribute retrieval with combined indexes. Comm. ACM, 13, 11 (Nov. 1970), 659-665.

NR-74 Nievergelt, J., and Reingold, E. M. Binary search trees of bounded balance. SIAM J. Computing 3, 1 (March 1973), 33-43.

_____, Binary search trees and file organization. Computing Surveys 6, 3 (Sept. 1974), 195-207.

Palermo-72
"A data base search problem," Fourth Interneth. Symposium on Computer and Information Science (COINS IV), Dec. 1972, Plenum Press, New York, 1972.

Palermo-75
"Performance evaluation of expressions in a relational algebra," Proc. ACM Pacific 75 Regional Conf., April 1975," ACM, New York, 1975, pp. 44-49.

Rothnie-74
"An approach to implementing a relational data management system," Proc. ACM SIGMOD Workshop on Data Description, Access, and Control, May 1974," ACM, New York, 1974, pp. 277-294.

Rothnie-76
"Evaluating inverted retrieval expressions in a relational data base management system," Proc. AFIPS National Computer Conf., May 1976, Vol. 46, AFIPS Press, Montvale, N. J., 1976, pp. 417-425.

San Jose-76
Astrahan, M. M., et al., "System R: A relational approach to data base management," ACM Transactions on Data Base Management 1, No. 2, 97-137 (June 1976).

SC-75
Smith, J. M., and Chang, P. "Optimizing the performance of a relational algebra data base interface," Comm. ACM 18, 10 (Oct. 1975), pp. 568-579.

Toronto-75
Mylopoulos, J., et al., "A multi-level relational system," Proceedings of the AFIPS National Computer Conference, May 1975 44, 402-408, AFIPS Press, Montvale, N. J., (1975).

WY-76
Wong, E., and Youssefi, K. "Decomposition - A Strategy for Query Processing," ACM Transactions on Data Base Systems, Vol 1 #3, Sept. 1976, pp. 223-241.

8

DTIC